34th International Conference on Concurrency Theory

CONCUR 2023, September 18–23, 2023, Antwerp, Belgium

Edited by

Guillermo A. Pérez
Jean-François Raskin

LIPICS

LIPIcs – Vol. 279 – CONCUR 2023 www.dagstuhl.de/lipics

Editors

Guillermo A. Pérez
University of Antwerp, Belgium
guillermo.perez@uantwerpen.be

Jean-François Raskin
Université libre de Bruxelles, Belgium
jraskin@ulb.ac.be

ACM Classification 2012
Theory of computation → Concurrency

ISBN 978-3-95977-299-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-299-0.

Publication date
September, 2023

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.CONCUR.2023.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Invited Paper

Invited Talk

Regular Papers

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

Contained within this volume are the peer-reviewed contributions accepted for the 34th International Conference on Concurrency Theory (CONCUR), held in 2023. CONCUR serves as an annual scientific forum for researchers, developers, and students working to expand the field of concurrency theory and its applications. In 2023, the University of Antwerp played host to CONCUR, arranging it alongside QEST 2023, FORMATS 2023, and FMICS 2023 as part of CONFEST 2023, which also featured several workshops one day before and one day after the main conferences.

For CONCUR 2023, we received 98 submissions and accepted 37 for presentation at the conference. The high standard of many submissions meant the acceptance criteria were stringent. We are grateful for the hard work of our program committee who produced 378 reviews with the help of 186 external expert reviewers. Their insights guided us in choosing a diverse set of papers after lively discussions following a rebuttal phase offered to the authors.

We wish to thank the authors for considering the feedback from our reviewers and submitting their revised work to the CONCUR 2023 proceedings. We are confident that the selected papers, due to their high quality, will give rise to interesting presentations and scientifically interesting discussions during the conference.

We are also proud that several well-respected scientists have agreed to deliver invited talks at the conference: Prof. Ahmed Bouajjani, Paris Diderot University, France, Prof. Joost-Pieter Katoen, RWTH Aachen, Germany (joint with all conferences), Prof. Nicolas Markey, University of Rennes, France (joint with FORMATS), Prof. Frans A. Oliehoek, TU Delft, Netherlands (joint with QEST), Prof. David Parker, Oxford University, UK (joint with QEST, FORMATS), Prof. Jaco van de Pol, Aarhus University, Denmark (joint with FORMATS, FMICS), and Prof. Anna Slobodova, Intel, USA (joint with FMICS)

In 2020, CONCUR and the IFIP WG 1.8 on Concurrency Theory initiated the test-of-time award to honor significant contributions to Concurrency Theory that were published at CONCUR. This year's award goes to Vincent Danos and Jean Krivine for their work "Reversible Communicating Systems," published in CONCUR 2004.

We address our thanks to the University of Antwerp for its assistance with CONCUR 2023 and CONFEST 2023, as well as to our sponsors, the Research Foundation – Flanders (FWO) and the Fund for Scientific Research (F.R.S.–FNRS).

Lastly, the proceedings of CONCUR 2023 are freely available through the LIPIcs series. We are grateful to the authors of the CONCUR 2023 papers, the participants of the conference, and the student volunteers from the University of Antwerp and the Université libre de Bruxelles for their contribution to making CONCUR 2023 a success.

CONCUR Test-Of-Time Award 2023

Bengt Jonsson
Department of Information Technology, Uppsala University, Sweden

Marta Kwiatkowska
Department of Computer Science, University of Oxford, UK

Igor Walukiewicz
CNRS, University of Bordeaux, France

Abstract

This short article recaps the purpose of the CONCUR Test-of-Time Award and presents the paper that received the Award in 2023.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases CONCUR Test-of-Time Award

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.1

Category Invited Paper

Acknowledgements We thank Javier Esparza (chair of the CONCUR Steering Committee) and Pedro d'Argenio (chair of the IFIP Working Group 1.8 on Concurrency Theory), and Guillermo A. Perez and Jean-Francois Raskin (chairs of the CONCUR 2023 Programme Committee) for their assistance throughout our work as a jury for this year's award.

1 Introduction

The CONCUR Test-of-Time Award was established in 2020 by the Steering Committee of the CONCUR conference and by the IFIP Working Group 1.8 on Concurrency Theory. Its purpose is to recognise important achievements in Concurrency Theory that were published at CONCUR and have stood the test of time. At its normal pace, starting from 2024, the CONCUR Test-of-Time Award will be attributed every other year, during the CONCUR conference, to one or two papers published in the 4-year period from 20 to 17 years earlier. In the transient period from 2020 to 2023, on the other hand, two such awards are attributed every year, in order to catch up with papers published in the first fifteen years of the conference, namely between 1990 and 2004. At CONCUR 2020 two awards were given, each rewarding two papers published in the period 1990–1995. Similarly, at CONCUR 2021 two awards were given, each rewarding two papers published in the period 1994–1999. At CONCUR 2022, four awards were given, two for the period 1998–2001 and another two for 2000-2003. We had the honour to serve as members of the fourth CONCUR Test-of-Time Award Jury. All papers published at CONCUR in the period 2002-2005 were eligible. After agreeing a shortlist of candidate papers and discussing their relative merits and infuence on the CONCUR research community and beyond, we selected the paper described below for the Award, out of a number of excellent candidates. The presentation of the Award will take place during CONCUR 2023, the 34th edition of the CONCUR conference, which is co-chaired by Guillermo A. Pérez and Jean-François Raskin, and will be held in Antwerp.

2 The Award Winning Contribution

For the period 2002–2005 the jury has chosen to award the paper:

Vincent Danos and Jean Krivine
Reversible Communicating Systems, published in CONCUR 2004
`https://doi.org/10.1007/978-3-540-28644-8_19`

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 1; pp. 1:1–1:2
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This paper represents the first exploration of the reversibility of concurrent computation within process algebra. The notion of reversible computation expands the conventional forward computation by incorporating the ability to roll back a computation. The roots of this concept can be traced back to the 1970s, where it was studied by Landauer and Bennett in the context of thermodynamics and Turing machines. They established that any deterministic computation could be simulated by a logically reversible Turing machine.

The challenge in applying reversibility to concurrent systems arises from the fact that actions are not linearly organized by execution time but are partially ordered by a causal relationship. The authors put forward the fundamental notion of causally-consistent reversibility capturing the concept that an action can only be undone if all its subsequent effects have been reversed. The introduced notion has direct applicability to reversibility in distributed settings.

This paper has since served as a source of inspiration, either directly or indirectly, for numerous studies on reversible concurrent systems modelled through (higher-order) process algebras, Petri nets, event structures, as well as reversible logic circuits made of DNA. The principle of reversibility has a wide range of applications in distributed systems, including debugging, rollback, and error recovery. These applications will undoubtedly continue to benefit from the pioneering and elegant formalization introduced by Danos and Krivine.

3 Concluding Remarks

Interview with the award recipients, which provides information on the historical context that led them to develop their award-winning work and on their research philosophy, has been conducted by Marta Kwiatkowska with the help of the jury members. The interview is accessible on the award's webpage `https://www.uantwerpen.be/en/conferences/confest-2023/concur/awards/`.

On Verifying Concurrent Programs Under Weakly Consistent Models

Ahmed Bouajjani
Université Paris Cité, CNRS, IRIF, France

Abstract

Developing correct and performant concurrent systems is a major challenge. When programming an application using a memory system, a natural expectation would be that each memory update is immediately visible to all concurrent threads (which corresponds to strong consistency). However, for performance reasons, only weaker guarantees can be ensured by memory systems, defined by what sets of updates can be made visible to each thread at any moment, and by the order in which they are made visible. The conditions on the visibility order guaranteed by a memory system corresponds to its memory consistency model. Weak consistency models admit complex and unintuitive behaviors, which makes the task of application programmers extremely hard. It is therefore important to determine an adequate level of consistency for each given application: a level that is weak enough to ensure performance, but also strong enough to ensure correctness of the application behaviors. This leads to the consideration of several important verification problems:

- the correctness of an application program running over a weak consistency model;
- the robustness of an application program w.r.t. consistency weakening;
- the fact that an implementation of a system (memory, storage system) guarantees a given (weak) consistency model.

The talk gives a broad presentation of these issues and some results in this research area. The talk is based on several joint works with students and colleagues during the last few years.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Concurrent programs, weakly consistent models

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.2

Category Invited Talk

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 2; pp. 2:1–2:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Reachability and Bounded Emptiness Problems of Constraint Automata with Prefix, Suffix and Infix

Jakub Michaliszyn
University of Wrocław, Poland

Jan Otop
University of Wrocław, Poland

Piotr Wieczorek
University of Wrocław, Poland

Abstract

We study constraint automata, which are finite-state automata over infinite alphabets consisting of tuples of words. A constraint automaton can compare the words of the consecutive tuples using Boolean combinations of the relations prefix, suffix, infix and equality.

First, we show that the reachability problem of such automata is PSPACE-complete. Second, we study automata over infinite sequences with Büchi conditions. We show that the problem: given a constraint automaton, is there a bound B and a sequence of tuples of words of length bounded by B, which is accepted by the automaton, is also PSPACE-complete. These results contribute towards solving the long-standing open problem of the decidability of the emptiness problem for constraint automata, in which the words can have arbitrary lengths.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases constraint automata, emptiness problem

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.3

Funding This work was supported by the National Science Centre (NCN), Poland under grant 2020/39/B/ST6/00521.

1 Introduction

Logics and automata over *data values*, i.e., values from an *infinite* domain, have applications in formal verification, automated reasoning, databases and others [7, 6, 4, 10, 2, 1, 19, 22].

A notable example of a formalism for data values is *constraint linear temporal logic (CLTL)* [7, 6, 4]. CLTL formulas are defined w.r.t. a relational structure, e.g. $(\mathbb{N}, =)$ or $(\mathbb{Z}, <)$; the variables in formulas range over the domain of the structure. *Constraints* are atomic formulas defined using variables and symbols from the structure. CLTL formulas combine such constraints with LTL modalities and Boolean connectives, i.e., CLTL is LTL in which propositional variables are replaced with constraints. Every CLTL formula defines a *data language*, which is a set of infinite sequences of data values.

Constraint automata is the automata-based formalism accompanying CLTL [5, 11, 7]. Again, they are parameterized by a structure, which can have an infinite domain. A constraint automaton is a Büchi automaton, in which transitions are labeled with constraints; an automaton can take a transition only if the current and the next data values satisfy the constraint labeling this transition. The satisfiability problem for CLTL can be reduced to the non-emptiness of constraint automata as in the LTL case. Furthermore, constraint automata can use the equality constraints to store a data value for future use, which makes them closely related to *register automata* [14, 22, 8].

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 3; pp. 3:1–3:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are several results for CLTL and constraint automata for specific structures (see survey [9]) but also for some unified classes, like linear orders [26, 16, 4, 3, 10].

We are interested in data values being strings (A^*) over a finite alphabet A. The structures, considered in the context of strings, may be equipped with basic relations on strings like *prefix*, *suffix* or *subsequence*. Motivations come from the fact that although temporal logics with constraints over integers can be helpful in formal analysis of programs with counters, in order to analyse pushdown systems we need constraints over strings and the prefix relation. The most typical of prefix and suffix usage is for queues: adding to a queue (represented as a word) corresponds to stating that the old word is a prefix of the new one, whereas removing from a queue corresponds to stating that the new word is a suffix of the old one, so both prefix and suffix are important (and infix can be expressed as a combination of prefix and suffix). The reachability problem for queue automata (a PDA with a queue instead of a stack) is undecidable, therefore analyzing systems with a queue is a challenging task.

There has been a large body of work on various problems involving strings especially for multiple variants of first-order logic (FO) [15, 12, 18]. These results have implications for CLTL and constraint automata as pointed out in [23]. The undecidability of the satisfiability problem for CLTL over structure $\mathbb{A} = (A^*, \leq_{\text{sub}}, (= w)_{w \in \Sigma^*})$, where A is a finite alphabet and $\leq_{\text{sub}}$ is the subsequence order, follows from undecidability of the satisfiability problem for Σ_1-fragment of FO logic over $\mathbb{A}$ [12].

The satisfiability problem for CLTL over $(A^*, \leq_{\text{p}}, =, (= w)_{w \in \Sigma^*})$, where $\leq_{\text{p}}$ is prefix order, is PSpace-complete [6]. Note that words with the prefix order alone form the structure isomorphic to an infinite tree with descendant/ancestor relations. However, it was shown in [3] that the known unified technique, involving the "existence of homomorphisms is decidable"-property, for satisfiability results of branching-time logics (like CTL* or ECTL*) [4] with integer constraints cannot be used to resolve the satisfiability status of temporal logics with constraints over trees. This in turn shows the difficulty of the result from [6]. In the automata approach the emptiness problem for constraint automata is PSpace-complete when the relation is the infinitely branching infinite order tree [16]. Because of the symmetry, the same complexity follows in the case when prefix order is replaced with suffix order.

Once the satisfiability for CLTL with the prefix (or the suffix) order alone is answered, a natural question arises: what happens if we study CLTL over the structure A^* equipped with both of them? This question has been asked by Demri and Deters [6]. Having both prefix and suffix allows for checking properties depending on both ends of strings like in Example 2 provided at the end of Section 2.2. In this work we also explicitly include the infix relation as well as a form of negation for each of the three relations (i.e., *incomparable w.r.t. prefix* or *suffix* or *infix* respectively). Although infix alone is definable with prefix and suffix using an additional variable, it is not the case for its negation. It is an important part of our results.

Peteler and Quaas [23] studied the emptiness problem for constraint automata over the prefix and the suffix orders. They exemplified that FO logic with the prefix order alone is decidable [25] while FO logic with the prefix order and the suffix order is undecidable (this follows from the undecidability result for the FO theory for the substring (infix) orders [17], and the fact that the substring order is FO-definable using prefix and suffix). On the positive side, as noted in [6], the Σ_1-fragment of FO logic is decidable for finite strings over a finite alphabet. The proof uses an algorithm based on the word equation approach [24, 21, 13].

In the same paper, Peteler and Quaas proved that it is decidable in NL when the automaton uses only a single variable that ranges over finite strings. The strings can be over a finite or countably infinite alphabet. Their proof proceeds by reduction to reachability queries on the finite graph underlying the automaton. They show that their technique works in the presence of equality tests with the empty string (similar to a zero test in one-counter automata). The result implies PSPACE-completeness of the satisfiability problem for `CLTL` over a single variable. In contrast, in our work we study the emptiness problem for constraint automata that can use many variables over finite strings, but the string lengths are bounded.

Our contribution

We study constraint automata with the *prefix*, *infix*, *suffix* and equality (=) relations. Our crucial technical contribution is Lemma 4 in Section 3, which provides a model-theoretic characterisation of the satisfiability of a maximal set of constraints expressed using the prefix, suffix and infix relations (without constants) over some (countable, possibly infinite) set of variables. It says that such a set of constraints is satisfiable if variables with the prefix (resp., the suffix) relation form a forest, and variables with the infix relation form a finitary partial order, i.e., every variable has finitely many predecessors.

We employ this lemma to show that the reachability problem of constraint automata is PSPACE-complete. To do so, we first introduce *type-tracking* automata, which store in the state the type of the processed input tuple, i.e., the information on the relation between all words in the input tuple. In such automata every path corresponds to some (pre)run over some sequence. For a constraint automaton, the type-tracking automaton may have exponential size, but it can be constructed on-the-fly and there is no need to store it in memory. Therefore, the reachability problem for unrestricted constraint automata can be solved in PSPACE. We also prove the matching lower bound.

For constraint automata over infinite sequences with Büchi conditions, we define the *bounded emptiness problem* as follows: given a constraint automaton, is there a bound B such that some sequence accepted by the automaton and all words in all tuples of that sequence have length bounded by B? We show that this problem is PSPACE-complete as well. To do so, we first prove that if there is a bounded sequence accepted by the automaton, there is an ultimately-periodic sequence accepted by the automaton. Next, we establish a condition for a cycle in such an ultimately-periodic sequence. Finally, we show how to check the existence of such a cycle in polynomial space. The corresponding lower bound can be obtained from the lower bound for the reachability problem.

2 Preliminaries

2.1 Relations and constraints

We assume Σ to be a finite *alphabet*, whose elements are *letters*. A *word* is a finite sequence of letters; ϵ denotes the empty word. For words w, v, the word wv is the concatenation of w and v. An n-tuple is a tuple consisting of n words. An n-sequence is a sequence of n-tuples. This is illustrated in Figure 1.

We say that a word w is

- a (strict) *prefix* of v, denoted as $w \sqsubset_P v$ if there is a non-empty word t such that $wt = v$;
- a (strict) *suffix* of v, denoted as $w \sqsubset_S v$, if there is a non-empty word t such that $tw = v$;
- a (strict) *infix* of v, denoted as $w \sqsubset_I v$ if there are words t, t', at least one of them non-empty, such that $twt' = v$.

We say that two words v, w are *incomparable with respect to the prefix order*, denoted as $v \perp_P w$ if none of the following holds: $v = w$, $w \sqsubset_P v$, $v \sqsubset_P w$. The $v \perp_S w$ and $v \perp_I w$ relations for suffix and infix are defined in a similar way.

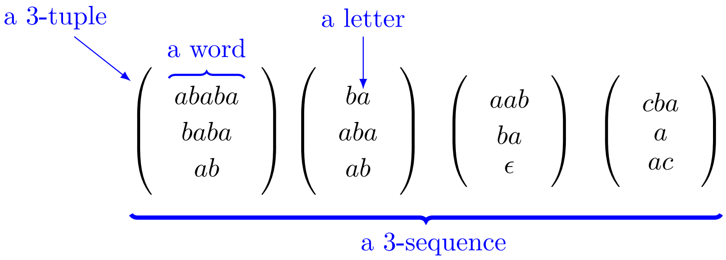


Figure 1 An example of a 3-sequence.

An *n-constraint* (over $\{x_1, \ldots, x_n\}$) is a set consisting of atoms of the form $x \oplus y$, where $x, y \in \{x_1, \ldots, x_n\}$ are variables and $\oplus \in \{<_P, <_S, <_I, \perp_P, \perp_S, \perp_I, =\}$ is a relation symbol. Given an n-tuple of words $\vec{w} = (w_1, \ldots, w_n)$, we define the interpretation $\mathrm{I}_{\vec{w}}$ such that for each variable x_i we have $\mathrm{I}_{\vec{w}}(x_i) = w_i$, and for relational symbols we have $I(<_P) = \sqsubset_P$, $\mathrm{I}_{\vec{w}}(<_S) = \sqsubset_S$, $\mathrm{I}_{\vec{w}}(<_I) = \sqsubset_I$, $\mathrm{I}_{\vec{w}}(\perp_P) = \perp_P$, $\mathrm{I}_{\vec{w}}(\perp_S) = \perp_S$, $\mathrm{I}_{\vec{w}}(\perp_I) = \perp_I$, and $\mathrm{I}_{\vec{w}}(=)$ is $=$. An n-constraint γ is *satisfied* by $\vec{w} = (w_1, \ldots, w_n)$, denoted as $\vec{w} \models \gamma$, if for every atom $x \oplus y$ from γ the expression over words $I_{\vec{w}}(x \oplus y)$ is true. An n-constraint γ is *satisfiable* if there exists an n-tuple of words $\vec{w}$ satisfying γ.

▶ **Example 1.** Consider the 6-constraint $\gamma = \{x_3 = x'_3, x_1 \perp_P x'_1, x'_1 <_S x_1\}$ over variables $x_1, x_2, x_3, x'_1, x'_2, x'_3$. Let w_1, w_2, w_3, w_4 be the 3-sequence presented in Figure 1. We will write (w_i, w_j) to denote a 6-tuple of words containing first the words of w_i, and then the words of w_j.

Then, $(w_1, w_2) \models \gamma$ holds, because $ab = ab, ababa \perp_P ba$ and $ba <_S ababa$. On the other hand, $(w_2, w_3) \models \gamma$ does not hold because $ab \neq \epsilon$. ◀

An ordered set $(X, <)$ is *finitary* if for every $t \in X$, the set $\{s \in X \mid s < t\}$ is finite. An ordered set $(X, <)$ is a (finitary) *tree* if for every $t \in X$, the set $\{s \in X \mid s < t\}$ is finite and totally ordered w.r.t. $<$. A (finitary) *forest* is a disjoint union of trees.

2.2 Constraint automata and their semantics

A (non-deterministic) *n-constraint automaton* $\mathcal{A} = (Q, Q_0, Q_F, \delta)$ is an automaton which processes tuples of finite words, i.e., $(\Sigma^*)^n$. Such an automaton consists of:

- A finite set of states Q and its subsets: initial states Q_0 and final states Q_F.
- A transition relation $\delta \subseteq Q \times \Gamma \times Q$, where Γ is the set of all satisfiable $2n$-constraints over $\{x_1, \ldots, x_n, x'_1, \ldots, x'_n\}$.

The *size* of an n-constraint automaton is the number of elements of Q and δ.

The semantics of n-constraint automata is defined over n-sequences. Intuitively, an automaton starts in an initial state with the first n-tuple, and then changes the state according to the following n-tuples and the transition relation. In transitions, unprimed variables $x_1, \ldots, x_n$ are interpreted as words at the origin and the primed variables are interpreted as words at the destination. The formal description follows below.

We will consider two semantics of constraint automata: over finite and infinite sequences. Let $\mathcal{A} = (Q, Q_0, Q_F, \delta)$ be an n-constraint automaton.

Constraint automata over finite sequences. A *partial run* of $\mathcal{A}$ over a finite sequence $w_0, \ldots, w_m$, with $m \geq 1$, is a sequence of states $q_0, \ldots, q_m$ where for each $i < m$ there is γ such that $(q_i, \gamma, q_{i+1}) \in \delta$ and $(w_i, w_{i+1}) \models \gamma$. A *run* is a special case of a partial run that starts in an initial state, i.e., $q_0 \in Q_0$. A run is *accepting* if the last state $q_m \in Q_F$.

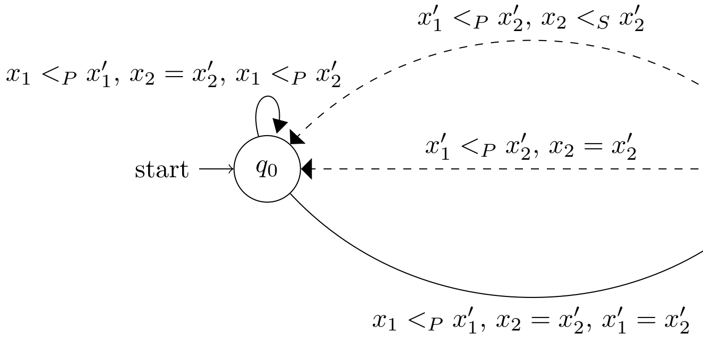

$x'_1 <_P x'_2,\ x_2 <_S x'_2$
$x_1 <_P x'_1,\ x_2 = x'_2,\ x_1 <_P x'_2$
start
q_0
$x'_1 <_P x'_2,\ x_2 = x'_2$
q_a
$x_1 <_P x'_1,\ x_2 = x'_2,\ x'_1 = x'_2$

Figure 2 An example of a 2-constraint automaton.

Constraint automata over infinite sequences. A *partial run* of $\mathcal{A}$ over an infinite sequence $w_0, w_1, \ldots$ is an infinite sequence of states $q_0, q_1, \ldots$ such that every finite prefix is a partial run over the corresponding prefix of $w_0, w_1, \ldots$. As before, a partial run is a *run* if it starts in an initial state. An infinite run is *accepting* if there is a state $q \in Q_F$ such that for infinitely many j we have $q_j = q$ (Büchi condition).

In both cases, we say that the automaton accepts a sequence s if there is an accepting run on it. The *language* of an automaton is the set of sequences it accepts. Two automata are equivalent if their languages are the same.

▶ **Example 2.** Consider the 2-constraint automaton depicted at Figure 2 without the dashed edges. This automaton, considered over finite sequences, accepts sequences $(w_1, v_1) \ldots (w_s, v_s)$ such that $w_s = v_s$, for all i, j we have $v_i = v_j$, for all $i < s$ we have $w_i <_P v_i$, $w_i <_P w_{i+1}$. This can model a process that starts from some list of tasks to accomplish (in a specific order) v and maintains the current list of finished tasks as w. The automaton accepts once the list is completed.

The automaton depicted in Figure 2 with the dashed edges can be considered on infinite sequences. In this case, the procedure described above is repeated infinitely often: once a current list is completed, the process starts over with a new list; in this example, the new list is the previous list possibly extended with new tasks at the beginning. The "new list" can be empty or non-empty, as no constraints check whether a word is empty. Note also how we have used nondeterminism to express the disjunction $x_2 = x'_2$ or $x_2 <_S x'_2$ during the return.

2.3 Decision problems

The *emptiness* problem for constraint automata over finite (resp., infinite) sequences is the question: given a n-constraint automaton, is there a finite (resp., an infinite) n-sequence accepted by this automaton?

For finite sequences, we consider a (slightly) more general question, that of *reachability*: given an n-constraint automaton $\mathcal{A}$ and its two states s, t, is there a finite n-sequence and a partial run over that sequence starting in s and ending in t? Even though the reachability problem and the emptiness problem over finite sequences are mutually reducible, the reachability problem is often more convenient to apply.

We say that an infinite n-sequence σ is bounded if there is a bound B, such that in every n-tuple $\sigma[i]$, words have length bounded by B. Note that finite and ultimately periodic sequences are bounded. The *bounded emptiness* problem is as follows: given a n-constraint automaton $\mathcal{A}$, is there an infinite bounded n-sequence accepted by $\mathcal{A}$?

3 Constraints and their Satisfaction

In this section, we study satisfiability of n-constraints. First, we give a characterization of satisfiability of constraints based on the shape of constraints (Lemma 4), which gives insight into the expressive power of constraints. While n-constraints are over finitely many variables, the characterization of Lemma 4 holds for countable sets of variables and hence it can have applications beyond this paper.

Next, we show a Craig-interpolation type lemma, which states that for two maximal satisfiable constraints, if they are consistent on the common variables, then their union is satisfiable (Lemma 6). We use Lemma 6 to derive a local-to-global principle, which states that local consistency implies global consistency for constraints resulting from runs of constraint automata (Theorem 10).

3.1 Satisfiability of maximal constraints

Let γ be a constraint over a set of variables $V = \{x_1, x_2, \ldots\}$.

▶ **Definition 3.** *We say that γ is* maximal *if for every pair of different variables x, y, either $x = y$ is in γ or for every $\rho \in \{P, S, I\}$ we have one of the following $x <_\rho y$, $y <_\rho x$ or $x \perp_\rho x'$ belongs to γ.*

Consider a maximal constraint γ. First, observe that we can eliminate equality constraints easily. Let E be the least equivalence relation on V containing all pairs (x, x') such that $x = x'$ occurs in γ. For each equivalence class C of E we pick the least i such that $x_i \in C$ and substitute all $y \in C$ with x_i. Let γ' be the resulting constraint, which we call the *equality-reduced* γ. If γ' has a conflicting pair of constraints (e.g. $y <_P y'$ and $y \perp_P y'$), then γ' is unsatisfiable as well as γ. Otherwise, if there is no such pair then γ' is a maximal constraint and it is satisfiable if and only if γ is.

Assume, without loss of generality, that γ is maximal and without equality constraints. We study three graphs over V: (V, P_γ), (V, S_γ), and (V, I_γ), which are obtained from γ by stating atomic constraints from γ as edges, i.e., we define $P_\gamma, S_\gamma, I_\gamma$ over V^2 such that for all $x, x' \in V$ we have $x P_\gamma x'$ (resp., $x S_\gamma x'$ or $x I_\gamma x'$) if and only if $x <_P x' \in \gamma$ (resp., $x <_S x' \in \gamma$ or $x <_I x' \in \gamma$).

Assume that γ is satisfiable, and it is satisfied by (possibly infinite) $\vec{w}$. Since γ is maximal and has no equality constraints, words in $\vec{w}$ are pairwise distinct. First, observe that graphs (V, P_γ) and (V, S_γ) are forests (union of disjoint trees). Indeed, a set of (pairwise distinct) words ordered by the prefix relation $\sqsubset_P$ is a forest, and hence (V, P_γ) is a forest. Similarly, (V, S_γ) is a forest as well. Second, observe that (V, I_γ) is an ordered set such that every element has finitely many predecessors, i.e., for $v \in V$ the set $A_v = \{u \in V \mid u I_\gamma v\}$ is finite. Indeed, $w \sqsubset_I w'$ implies that $|w| < |w'|$ and hence there are no infinite descending chains. Finally, I_γ contains P_γ and S_γ as every prefix (resp., suffix) is an infix as well.

Interestingly, these properties are in fact sufficient for γ to be satisfiable.

▶ **Lemma 4.** *Let γ be a maximal constraint without equality over the set of variables V. Then, γ is satisfiable if and only if (V, P_γ) and (V, S_γ) are forests, (V, I_γ) is a finitary ordered set, and I_γ contains P_γ and S_γ.*

We sketch the proof of the remaining implication, that the above conditions imply satisfiability of γ. We construct an assignment satisfying γ as follows. We first consider a possibly infinite set $\Gamma = \{a_v \mid v \in V\}$ as the alphabet; we reduce the obtained assignment later.

For all minimal elements v in (V, I_γ), we assign the letter a_v to v. Then, inductively, for an I_γ-minimal unassigned v, we proceed as follows. Our assumption is that for every pair of different variables x, y such that $xI_\gamma v$ and $yI_\gamma v$ all the constraints in γ involving both x and y are satisfied.

Since (V, P_γ) is a forest, either v has a unique P_γ-predecessor u_P, to which a word w_P is assigned, or v is P_γ-minimal and we put $w_P = \epsilon$. Similarly, either v has a unique S_γ-predecessor u_S, to which a word w_S is assigned, or v is S_γ-minimal and we put $w_S = \epsilon$. Finally, let A be the set of all u such that $uI_\gamma v$. Since (V, I_γ) is a well partial order, the set A is finite. Let $u[1], \ldots, u[k]$ be all words in A and $w' = w_{u[1]} \ldots w_{u[k]}$ be the concatenation of the words that have already been assigned to $u[1], \ldots, u[k]$. Then, we assign with v the word $w_v = w_P a_v w' a_v w_S$. Note that this is the first time the letter a_v is used.

Observe that for every $u \in V$, if $u <_P v \in \gamma$, then u has already been assigned with a word. Indeed, $uP_\gamma v$ and due to $P_\gamma \subseteq I_\gamma$ we have $uI_\gamma v$, and hence u has already been considered. It follows that all constraints $u <_P v \in \gamma$ are satisfied by the assignment. Similarly, all constraints $u <_S v \in \gamma$ and $u <_I v \in \gamma$ are satisfied by the assignment.

Now, observe that exactly these positive constraints are satisfied. First, consider, for an already assigned u that is different from v, the constraint $u <_I v \notin \gamma$. Recall that w_v is the word assigned to v and let w_u be the word assigned to u. We show that $w_u \perp_I w_v$ and hence the constraint $u \perp_I v$, which has to belong to γ due to maximality, is satisfied. Indeed, observe that I_γ contains P_γ and S_γ, and w_P, w' and w_S contain only letters a_x such that $xI_\gamma v$. Since $u <_I v \notin \gamma$ implies $uI_\gamma v$ does not hold, we get that w_v does not contain a_u. Moreover, w_u does not contain a_v and hence $w_u \perp_I w_v$.

Second, consider $u <_P v \notin \gamma$. We show $w_u \perp_P w_v$. If $uI_\gamma v$ does not hold, then $w_u \perp_I w_v$ and in particular $w_u \perp_P w_v$. Therefore, $uI_\gamma v$ holds. Assume towards contradiction $w_u \sqsubset_P w_v$. We know that w_u does not contain a_v because u was assigned before v. Therefore, if $w_u \sqsubset_P w_v$ then w_u is either equal to or is a prefix of w_v. In this case, w_v has to be non-empty. Recall also that it is the word w_x assigned to x, the P_γ-predecessor of v. Note, however that w_u is not equal to $w_P = w_x$ as all the words in the constructed substitution are different. Moreover, if $w_u \sqsubset_P w_x$ then due to the induction hypothesis and because of $P_\gamma \subseteq I_\gamma$ we have $u <_P x \in \gamma$. Therefore $u <_P v \in \gamma$, a contradiction. Thus, $w_u \perp_P w_v$ and $u \perp_P v$ is satisfied.

Similarly, if $u <_S v \notin \gamma$, then $u \perp_S v$ is satisfied. As a consequence, the constructed substitution over $\Gamma = \{a_v \mid v \in V\}$ satisfies γ.

Finally, we can transform the variable assignment over the infinite alphabet Γ to a satisfying assignment over any Σ with at least two letters. We take two distinct $b, c \in \Sigma$ and enumerate $a_1, a_2, \ldots$ the set Γ. Next, we apply to each $a_i \in \Gamma$ in the assignment the transformation $a_i \mapsto bc^i b$. One can easily check, that this transformation preserves prefixes, suffixes and infixes, and hence it is an assignment over Σ satisfying γ.

Observe that having a finite maximal constraint γ, we can eliminate equality in polynomial time, and then check the conditions of Lemma 4 in polynomial time as well. As a consequence we have:

▶ **Lemma 5.** *The satisfiability problem for maximal constraints can be solved in polynomial time.*

3.2 Joining constraints

We now prove the second crucial lemma that says that whenever we have two maximal satisfiable sets of constraints, if the sets agree on the constraints regarding the common variables, then the union of these sets is satisfiable.

▶ **Lemma 6.** *Let γ_1, γ_2 be maximal satisfiable constraints over variables X_1, X_2 respectively. If γ_1 and γ_2 restricted to $X_1 \cap X_2$ coincide, then $\gamma_1 \cup \gamma_2$ is satisfiable.*

First, observe that it suffices to show the lemma in the special case of $X_1 = X \cup \{x\}$ and $X_2 = X \cup \{z\}$, i.e., X_1 and X_2 differ in two variables.

▶ **Lemma 7.** *Let γ_1, γ_2 be maximal satisfiable constraints over variables $X \cup \{x\}, X \cup \{z\}$ respectively. If γ_1 and γ_2 restricted to X coincide, then $\gamma_1 \cup \gamma_2$ is satisfiable.*

Proof. The proof strategy is to define a maximal γ over $V = X \cup \{x, z\}$ such that $\gamma_1 \cup \gamma_2 \subset \gamma$, (V, P_γ) and (V, S_γ) are forests, (V, I_γ) is a finitary ordered set, and I_γ contains P_γ and S_γ. Then, Lemma 4 delivers the satisfiability of γ, and therefore $\gamma_1 \cup \gamma_2$. Since γ_1 and γ_2 are maximal, we only need to define the relation between x and z in γ.

First, we check whether some of the relations follow from transitivity. More precisely, we define γ^T as the least constraint that subsumes γ_1 and γ_2 and such that all the relations among $\{<_P, <_S, <_I\}$ are transitively closed in γ^T.

We can show that the relations $P_{\gamma^T}, S_{\gamma^T}$ and I_{γ^T} in γ^T are partial orders. The reflexivity holds trivially and transitivity follows from the definition. To see that the relations are antisymmetric, observe that the transitive closure only defines relations between x and z, as γ_1 and γ_2 are maximal and satisfiable and hence transitively closed. we discuss the case of I_{γ^T} here, the others are analogous. Assume towards contradiction that γ^T contains $x <_I z$ and $z <_I x$; then there are $v, v' \in X$ such that in $\gamma_1 \cup \gamma_2$ we have $x <_I v$, $v <_I z$ and $z <_I v', v' <_I x$. However, since γ_1 is maximal, $<_I$ is transitive and hence $v' <_I v$ is in γ_1. Similarly, $v <_I v'$ is in γ_2. Since γ_1 and γ_2 restricted to X coincide, we have both $v <_I v'$ and $v' <_I v$ belong to γ_1 and γ_2 and hence they are not satisfiable.

It is possible that $(X \cup \{x, z\}, P_{\gamma^T})$ is not a forest. This happens when both x and z are prefixes of some variable y, but the prefix order between x and z is not set. We fix this order as follows. If a constraint determines that $x <_I z$ or $z <_I x$, then we set $x <_P z$ or $z <_P x$ accordingly. Otherwise, we set the order in an arbitrary way. The same reasoning applies to the suffix order.

More precisely, we construct the set γ as an extension of γ^T in the following way. For each $R \in \{P, S\}$, if there is $y \in X$ such that $x \leq_R y$ and $z \leq_R y$, then we add to γ:

- $x \leq_R z$ if $x \leq_I z \in \gamma^T$
- $z \leq_R x$ and $z \leq_I x$ otherwise.

If there is no y such that $x \leq_R y$ and $z \leq_R y$, and the R-relation between x and z is not defined, we set x and z to be R incomparable in γ. This concludes the construction of γ.

This construction guarantees that I_γ contains P_γ and S_γ, and P_γ, S_γ, I_γ are partial orders. The last step ensures that (V, P_γ) and (V, S_γ) are forests. To see that (V, I_γ) is finitary in γ, observe that (V, I_γ) was finitary is γ_1 and γ_2. Thus, every variable in $X \cup \{x\}$ has in γ finitely many predecessors (at most one more than it has in γ_2), and the same holds for $X \cup \{z\}$. ◀

Lemma 6 follows from the above lemma using inductive reasoning.

Lemma 6 shows that a finite union of finite satisfiable constraints is satisfiable. This does not translate to the infinite union case; the following example shows a constraint that can be repeated any number of times, but not infinitely many times.

▶ **Example 8.** Consider a single-state 1-constraint automaton $\mathcal{A}$ with the state q that is both initial and accepting. The only transition is $(q, \{x_1' <_P x_1\}, q)$. This automaton can accept sequences of arbitrary length; it also has an infinite path $(q, q, q, \dots)$, but it does not accept any infinite sequence. This is because there is no infinite sequence of finite words such that each consecutive word is a prefix of the previous one.

3.3 Stratified constraints and their satisfaction

In this section, we connect general constraints with constraints that are derived from partial runs of n-constraint automata. One of the key differences is a natural structure of constraints resulting from partial runs; such constraints are local, which is captured by the following definition.

Consider a (partial) run π of an n-constraint automaton over some finite sequence σ and let $\gamma_1, \ldots, \gamma_k$ be the sequence of constraints along transitions of this run. First, for each j we relabel variables in γ_j in a way such that x_i becomes x_i^{j-1} and x_i' becomes x_i^j. We denote the resulting constraint by γ_j'. Second, the constraints $\gamma_1', \ldots, \gamma_k'$ can be extended to maximal satisfiable constraints $\gamma_1^t, \ldots, \gamma_k^t$, which agree on the common variables; it suffices to check the relations in the sequence σ. Consider γ_π to be the union of constraints $\gamma_1^t, \ldots, \gamma_k^t$. The constraints in γ_π are local, which is captured by the following definition of stratified constraints; there are constraints only between variables corresponding to the successive positions.

▶ **Definition 9.** *Given a natural number n and $k \in \mathbb{N}$, a* stratified (n,k)-constraint *γ is a constraint over the set of variables of the form x_i^j, where $i \in \{1, \ldots, n\}$ and $0 \leq j < k$, such that all the atoms $x_i^j \oplus x_{i'}^{j'}$ are such that $j - j' \in \{-1, 0, 1\}$. The* j-th layer *of a stratified (n,k)-constraint is the set of variables $x_1^j, \ldots, x_n^j$.*

The constraint γ_π is a stratified (n,k)-constraint. As it results from a (partial) run (the run π), it is satisfiable. However, satisfiability of γ_π follows also from a general principle.

We show a local-to-global principle, which states that for stratified constraints local consistency (subconstraints γ_j^t are satisfiable and agree on common variables) implies global consistency (i.e., γ_π is satisfiable.)

▶ **Theorem 10.** *Let γ be a stratified (n,k)-constraint such that $n, k \in \mathbb{N}$ and for every $0 \leq j < k-1$ the constraint γ restricted to layers $j, j+1$ is maximal and satisfiable. Then, the constraint γ is satisfiable.*

The proof of Theorem 10 follows by induction from Lemma 6. Consider a stratified $(n, k+1)$-constraint γ and let $\hat{\gamma}_1$ be the constraint obtained from γ by dropping the last layer. Since $\hat{\gamma}_1$ is a stratified (n,k)-constraint, assume that it is consistent. Let $\hat{\gamma}_2$ be the $(n,2)$-constraint consisting of the last two layers: k-th and $(k+1)$-th. Note that $\hat{\gamma}_2$ is maximal and satisfiable and the intersection of $\hat{\gamma}_1$ and $\hat{\gamma}_1$ is the k-th layer. Therefore, by Lemma 6 the constraint γ is satisfiable.

4 Reachability via type-tracking automata

We introduce a special type of n-constraint automata, called *type-tracking automata*, which keep track of the *type* (intuitively: what relations hold between the words of the tuple) of the current tuple in the states. While in n-constraint automata a path in the automaton, considered a labeled graph, may not correspond to a partial run, which involves satisfaction of constraints along the path, in type-tracking automata every finite path corresponds to a partial run over some sequence. This property is key in solving the reachability problem for n-constraint automata.

The *type* of an n-tuple $\vec{w}$ is the set of all non-trivial atomic n-constraints over $\{x_1, \ldots, x_n\}$ satisfied for $\vec{w}$. Observe that a constraint γ is a type if and only if it is maximal and satisfiable.

In the above, non-trivial atomic constraints are the constraints of the form $x \oplus y$ where x and y are different variables. For example, the type of the first 3-tuple of Figure 1 is the set containing the following atoms:

- $x_1 \perp_P x_2, x_2 \perp_P x_1, x_2 <_S x_1, x_2 <_I x_1$
- $x_3 <_P x_1, x_1 \perp_S x_3, x_3 \perp_S x_1, x_3 <_I x_1$
- $x_2 \perp_P x_3, x_3 \perp_P x_2, x_2 \perp_S x_3, x_3 \perp_S x_2, x_2 <_I x_3$

Let $\mathbb{T}$ be the set of all types (of some n-tuples). The set $\mathbb{T}$ is exponentially bounded in n and their members have size polynomial in n.

We now introduce a special version of n-constraint automata whose states carry information regarding the type of current n-tuple.

▶ **Definition 11.** *For an n-constraint automaton $\mathcal{A} = \langle Q, Q_0, Q_F, \delta\rangle$, the* type-tracking n-constraint automaton *$\mathcal{A}^{FT}$ resulting from $\mathcal{A}$ is the n-constraint automaton $\langle Q', Q'_0, Q'_F, \delta'\rangle$ such that:*

- *the states of $\mathcal{A}^{FT}$ are the pairs of a state of $\mathcal{A}$ and a type: $Q' = Q \times \mathbb{T}$, $Q'_0 = Q_0 \times \mathbb{T}$ and $Q'_F = Q_F \times \mathbb{T}$,*
- *δ' is the set of tuples $\langle (q_1, \gamma_1), \gamma', (q_2, \gamma_2)\rangle$, such that for some $\langle q_1, \gamma, q_2\rangle \in \delta$, the constraint γ' is a maximal consistent constraint that contains γ_1, γ_2 and γ.*

We say that a partial run $(q_0, \gamma_0), (q_1, \gamma_1), \ldots$ (finite or infinite) of a type-tracking n-constraint automaton $\mathcal{A}^{FT}$ over a sequence $\sigma = w_0, w_1, \ldots$ is *consistent* if for every $0 \leq i \leq |\sigma| - 1$ we have γ_i is the type of $\sigma[i]$. Observe that every partial run of an n-constraint automaton has the corresponding *consistent* run in the type-tracking automaton.

The main advantage of type-tracking n-constraint automata is that every path in a type-tracking automaton corresponds to some partial run, which is not the case for n-constraint automata in general.

▶ **Lemma 12.** *Let $\mathcal{A}$ be an n-constraint automaton and $\mathcal{A}^{FT}$ be its the type-tracking n-constraint automaton.*

1. *Every (finite or infinite) partial run in $\mathcal{A}$ over a sequence σ has a (unique) corresponding consistent partial run of $\mathcal{A}^{FT}$ over σ.*
2. *Every finite path in $\mathcal{A}^{FT}$ corresponds to a partial run of $\mathcal{A}^{FT}$ over some sequence σ.*

Proof. Property 1 follows from augmenting states of the partial runs with the types of the corresponding tuples of the given sequence. To see 2, observe that a path π of length k in the type-tracking automaton $\mathcal{A}^{FT}$ yields a stratified (n,k)-constraint such that any two successive layers are maximal and satisfiable. Thus, by Theorem 10 it is satisfiable and hence there is a sequence σ such that $\mathcal{A}^{FT}$ over σ has a consistent partial run corresponding to π. ◀

Type-tracking n-constraint automata are typically exponentially larger than their n-constraint counterparts. For example, consider a single state n-constraint automaton accepting all the sequences. Any corresponding type-tracking n-constraint automaton has to have at least as many states as there are types, so exponentially many.

The type-tracking n-constraint automata need not be explicitly stored. We can compute the states of type-tracking n-constraint automata *on the fly*. To do so, we employ the following result:

▶ **Lemma 13.** *For a given n-constraint automaton $\mathcal{A}$, the following problems can be solved in polynomial time.*

1. *Given (s,t), check whether (s,t) is a state of $\mathcal{A}^{FT}$.*
2. *Given (s_1,t_1), (s_2,t_2) and γ, check whether $((s_1,t_1), \gamma, (s_2,t_2))$ is a transition of $\mathcal{A}^{FT}$.*

Proof. To check (s,t) whether it is a state of $\mathcal{A}^{FT}$ it suffices to check whether s is a state of $\mathcal{A}$ and t is a type. The latter can be done in polynomial time as follows. Checking maximality is straightforward and for maximal constraints checking satisfiability can be done in polynomial time (Lemma 5).

Solving 2 amounts to checking maximality, satisfiability and containment, which can be done in polynomial time. ◀

Lemma 13 implies that graph-reachability in $\mathcal{A}^{FT}$ can be solved in polynomial space in $|\mathcal{A}|$.

▶ **Lemma 14.** *The problem: given an n-constraint automaton, its states q_1, q_2 and two types γ_1, γ_2, decide whether (q_2, γ_2) is path-reachable from (q_1, γ_1) in the type-tracking n-constraint automaton resulting from $\mathcal{A}$, is in* PSPACE.

We now show the upper bound for the reachability problem.

▶ **Theorem 15.** *The reachability problem for n-constraint automata is in* PSPACE.

This theorem follows from Lemma 12 and Lemma 14. To check the reachability from q_1 to q_2, the algorithm non-deterministically picks (recall that Savitch's Theorem proves that PSPACE=NPSPACE) two types γ_1, γ_2 and employs Lemma 14 to check if there is a path in the type-tracking automaton. By Lemma 12, such γ_1, γ_2 and a path exist if and only if there is a path from q_1 to q_2.

We show PSPACE-hardness of reachability in n-constraint automata. For $n > 0$, we say that a propositional formula ϕ over $2n$ variables *represents* a directed graph $G = (V, E)$, if V is the set of binary sequences of length n, and for all vertices $\vec{x}, \vec{y}$, we have $E(\vec{x}, \vec{y})$ if and only if $\phi(\vec{x}, \vec{y})$ is satisfied. The *reachability problem in succinct graphs* is defined as follows: given $n > 0$, a formula ϕ over $2n$ variables and two binary sequences $\vec{s}, \vec{t}$ of length n, decide whether $\vec{t}$ is reachable from $\vec{s}$ in the graph represented by φ. This problem is known to be PSPACE-complete [20].

We say that a propositional formula ϕ is in an *extended-DNF* if it is a disjunction of conjunctions of *literals* of the following three forms: p_i, $\neg p_i$ or $p_i \Leftrightarrow p_j$. Note that in the standard DNF, the equivalence is not allowed. It turns out that extended-DNF formulas are enough to make the reachability problem in succinct graphs PSPACE-complete.

▶ **Lemma 16.** *The reachability problem in graphs given by formulae in extended-DNF is* PSPACE*-complete.*

The proof follows from the fact that extended-DNF are sufficient to express property of being the successor configuration of polynomial-space Turing machine. The main idea is that the two consecutive configurations of a Turing machine differ only on a head position, a state of the Turing machine, and at most one tape cell; this can be expressed using a disjunction of polynomially many formulas. The remaining part of the configuration is the same, and this can be expressed in the conjunctions using $\Leftrightarrow$.

We now prove the lower bound for the n-constraint automata.

▶ **Theorem 17.** *The reachability problem for n-constraint automata is* PSPACE*-hard.*

Proof. Observe that formulae in extended-DNF can be encoded in an $(n+2)$-constraint automaton, and hence the reachability problem in succinct graphs reduces to the reachability problem in constraint automata with three states: q_0, q, q_F. To do so, we designate the last two elements w_{n+1}, w_{n+2} in each tuple to be different words (e.g. stating in the constraints $x_{n+1} \perp_I x_{n+2}$), which do not change along the run, and encode *true* and *false* in the propositional sense. Then, literals p_i, $\neg p_i$ and $p_i \Leftrightarrow p_j$ are respectively translated to constraints $x_i = x_{n+1}$, $x_i = x_{n+2}$, and $x_i = x_j$. The conjunction of literals can be stated in the constraints, and the disjunction can be encoded with non-determinism of the $(n+2)$-constraint automaton, i.e., the disjunction $d_1 \vee \ldots \vee d_k$ is translated to k transitions from q to itself each with the constraints resulting from d_i, which is a conjunction of literals. Finally, we set the first transition from q_0 to q to set the initial vertex in the reachability problem in graphs given by a propositional formula and one outgoing transition from q to q_F, which is possible only with the valuation of variables corresponding to the final vertex in the instance of the problem. ◀

As a direct consequence of Theorems 15 and 17 we have:

▶ **Corollary 18.** *The reachability problem for n-constraint automata is PSpace-complete.*

5 Checking emptiness over bounded words

In this section, we consider constraint automata over bounded sequences over finite alphabets. We establish PSpace-completeness of the emptiness problem for constraint automata restricted to bounded sequences. Notice that Example 8 shows that there is no straightforward counterpart of Lemma 12 for infinite runs.

We show that we can focus on ultimately periodic runs over an ultimately periodic sequences.

▶ **Lemma 19.** *An n-constraint automaton has an accepting run over some bounded infinite sequence if and only if it has an accepting ultimately periodic run over an ultimately periodic sequence.*

Proof. Observe that having a bound B, the set of B bounded words is finite and hence an n-constraint automaton $\mathcal{A}$ over B-bounded sequences can be considered as a Büchi-automaton. Therefore, if $\mathcal{A}$ accepts a B-bounded sequence, then it accepts an ultimately periodic B-bounded sequence. Clearly, every ultimately periodic sequence is bounded from some B. As a consequence, we can focus on ultimately periodic words. ◀

To decide whether there exists an ultimately periodic sequence $\sigma_0\sigma_1^\omega$ it suffices to decide the existence of an appropriate σ_0 and σ_1 almost independently. First, it is convenient to work with the type-tracking n-constraint automaton $\mathcal{A}^{FT}$ for $\mathcal{A}$, as every finite path there is realizable. Furthermore, there is a simple condition for a cycle, which can be iterated indefinitely. We discuss it in the following section.

5.1 Finding a cycle

The cycle c of $\mathcal{A}^{FT}$ defines a stratified (n,k)-constraint γ, which is satisfiable, i.e., it is a partial run over some sequence σ_1. We say that the cycle c is *iterable* if and only if it contains an accepting state and γ extended with constraints $x_1^0 = x_1^k, \ldots, x_n^0 = x_n^k$ (equality between corresponding variables in the first and the last layer) is still satisfiable. Observe, that if c is an iterable cycle, then c^ω is a partial run over σ_1, i.e., it is realizable.

▶ **Lemma 20.** *Let $\mathcal{A}$ be an n-constraint automaton and $\mathcal{A}^{FT}$ be its type-tracking n-constraint automaton. The automaton $\mathcal{A}$ has an accepting ultimately periodic run over an ultimately periodic sequence if and only if there exists a state s in $\mathcal{A}^{FT}$ such that*

- *(s,γ) is reachable from the initial state, and*
- *there exists an iterable cycle c from (s,γ) to itself.*

Observe that having a state (s, γ) in $\mathcal{A}^{FT}$, which can be non-deterministically picked, the first condition can be verified in PSpace due to Theorem 15. For the second condition, we can employ the following non-deterministic procedure, which works in polynomial space in $|\mathcal{A}|$, as follows.

▶ **Lemma 21.** *Given an n-constraint automaton $\mathcal{A}$ and a state s of its type-tracking n-constraint automaton $\mathcal{A}^{FT}$, one can decide in PSpace whether there is an iterable cycle in $\mathcal{A}^{FT}$ from s to itself.*

Proof. Our non-deterministic algorithm is similar to the standard on-the-fly reachability checking, but it requires additional information regarding the traversed path to ensure that the computed cycle is iterable. In particular, it ensures that an accepting state has been visited and verifies the relations between the initial and the final configuration, that may depend on the path.

The algorithm stores five objects: the initial state (s, γ_1), the current state (t, γ_2), number of steps k, the maximal constraint γ containing γ_1, γ_2 over the variables from γ_1 and γ_2, and a boolean value *Acc* stating whether an accepting state has been observed. We start with the state (s, γ_1) and initially $(s, \gamma_1) = (t, \gamma_2)$, $k = 0$, γ being the constraint describing two copies of γ_1 and the equality constraints between the corresponding variables, and *Acc* being true if s is accepting. Then, we compute the next value so that the following invariant holds:

inv (t, γ_2) is reachable from (s, γ_1) over some sequence of length k consistent with γ, i.e., there is a sequence σ of length k such that (a) there is a partial run (visiting an accepting state if *Acc* it true) over σ from (s, γ_1) to (s, γ_2), and (b) the constraint γ is consistent with the relations over $\sigma[1]$ and $\sigma[k]$.

We discuss how to maintain the invariant (inv). Assume that $(s, \gamma_1), (t, \gamma_2)$, k, γ and *Acc* are correct. Now, suppose that t' is some successor of t in $\mathcal{A}^{FT}$ and γ^+ is any maximal consistent constraint over variables from s, t, t'. The projection of γ^+ on the variables of s and t' satisfies the invariant. To see this, we apply Lemma 6 to $\hat{\gamma}_1$ being the constraint corresponding to the sequence σ and a partial run from s to t, and $\hat{\gamma}_2$ being γ^+. Both sets are consistent and they agree over the common variables, therefore their union is satisfiable and the satisfying sequence has length $k + 1$.

As a consequence, it suffices to execute this non-deterministic procedure until it reaches the state with $(s, \gamma_1), (s, \gamma_1)$, $k > 0$, $Acc = true$ and γ containing the equality constraint for the corresponding variables, in which case it accepts, or it works indefinitely, but it can be stopped after k exceeds the number of states of $\mathcal{A}^{FT}$ times the number of possible transitions, which is exponential in n. ◀

5.2 Solving the reachability problem

We can now conclude that solving reachability can be done in polynomial space.

▶ **Theorem 22.** *Checking whether there is a bounded infinite sequence accepted by a given constraint automaton $\mathcal{A}$ can be done in polynomial space in $|\mathcal{A}|$.*

The (non-deterministic) algorithm guesses a state s and a type γ such that (s, γ) is reachable from the initial state, and there exists an iterable cycle c from (s, γ) to itself. Verifying both properties was shown to be decidable in polynomial space. Since NPSpace= PSpace, the same can be done deterministically in polynomial space.

The matching lower bound follows from a straightforward reduction from the reachability problem.

▶ **Theorem 23.** *Checking whether there is a bounded infinite sequence accepted by a given constraint automaton is PSpace-complete.*

6 Conclusions

We have shown that the reachability problem and the non-emptiness over bounded sequences problem are PSPACE-complete for constraint automata. The proof works for constraint automata with the prefix, the suffix, the infix and the equality relations. The presented hardness proof requires only equality and negated equality, which can be expressed having non-strict order (prefix, infix or suffix). We believe that it can be adapted to the case of two relations: strict prefix and negated strict prefix (resp., suffix). This shows that the complexity follows from the number of variables.

The remaining open question is whether the (unrestricted) emptiness problem for constraint automata over infinite words is decidable. Lemma 4 gives us some insight. An important step towards this result would be to determine whether every non-empty constraint automaton has an accepting run (over some sequence) that is ultimately periodic. We discuss here an example demonstrating that it is not as straightforward as it may initially appear.

Consider a 3-constraint automaton $\mathcal{A}$ with a single state and a single transition. This transition is a conjunction of the following atoms:

- $x_1 = x_1'$
- $x_2 <_I x_2'$
- $x_3 \perp_I x_2$
- $x_3 <_I x_1$
- $x_3 <_I x_2'$

At first glance, it seems that the language of this automaton should be non-empty: x_1 is always the same, x_2 always increases, and x_3 is defined based on variables x_1 and x_2. To illustrate the issue, consider the following sequence:

$$\begin{pmatrix} abcde \\ x \\ a \end{pmatrix} \begin{pmatrix} abcde \\ ax \\ bc \end{pmatrix} \begin{pmatrix} abcde \\ axbc \\ bcd \end{pmatrix} \begin{pmatrix} abcde \\ axbcd \\ bcde \end{pmatrix}$$

Observe that this sequence is accepted by $\mathcal{A}$, but it cannot be extended in a way that maintains the acceptance. This is because the next value of x_2 must include all the infixes of x_1, which is contradictory with the conditions stating that x_3 is an infix of x_1 not contained in x_2. It can be easily checked that $\mathcal{A}$ does not accept any infinite sequence.

This example can be extended (by adding a lot of constraints to the only transition) in such a way that the only transition is maximal. In this case, there exist arbitrarily long sequences accepted by the automaton, where the types of all tuples and relations between all pairs of tuples in the same order are the same. Moreover, it can be done in a way that the lengths of x_2 and x_3 always increase (and x_1 remains unchanged). Despite this, there is no infinite sequence accepted by this automaton. This shows that formulating a pumping-lemma-esque argument in this context is elusive.

References

1 Mikołaj Bojańczyk. Atom book, September 2019. URL: `https://www.mimuw.edu.pl/~bojan/upload/main-10.pdf`.

2 Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011. doi:10.1145/1970398.1970403.

3 Claudia Carapelle, Shiguang Feng, Alexander Kartzow, and Markus Lohrey. Satisfiability of ECTL$_*$ with local tree constraints. *Theory Comput. Syst.*, 61(2):689–720, 2017. doi:10.1007/s00224-016-9724-y.

4 Claudia Carapelle, Alexander Kartzow, and Markus Lohrey. Satisfiability of ECTL$_*$ with constraints. *J.omput. Syst. Sci.*, 82(5):826–855, 2016. doi:10.1016/j.jcss.2016.02.002.

5 Karlis Cerans. Deciding properties of integral relational automata. In *Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings*, volume 820, pages 35–46. Springer, 1994. doi:10.1007/3-540-58201-0_56.

6 Stéphane Demri and Morgan Deters. Temporal logics on strings with prefix relation. *J. Log. Comput.*, 26(3):989–1017, 2016. doi:10.1093/logcom/exv028.

7 Stéphane Demri and Deepak D'Souza. An automata-theoretic approach to constraint LTL. *Inf. Comput.*, 205(3):380–415, 2007. doi:10.1016/j.ic.2006.09.006.

8 Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3):16:1–16:30, 2009. doi:10.1145/1507244.1507246.

9 Stéphane Demri and Karin Quaas. Concrete domains in logics: a survey. *ACM SIGLOG News*, 8(3):6–29, 2021. doi:10.1145/3477986.3477988.

10 Stéphane Demri and Karin Quaas. Constraint automata on infinite data trees: From CTL(Z)/CTL*(Z) to decision procedures, 2023. arXiv:2302.05327.

11 Régis Gascon. An automata-based approach for CTL∗ with constraints. In *Joint Proceedings of the 8th, 9th, and 10th International Workshops on Verification of Infinite-State Systems, INFINITY 2006 / 2007 / 2008*, volume 239 of *Electronic Notes in Theoretical Computer Science*, pages 193–211. Elsevier, 2009. doi:10.1016/j.entcs.2009.05.040.

12 Simon Halfon, Philippe Schnoebelen, and Georg Zetzsche. Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005141.

13 Artur Jeż. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, 2016. doi:10.1145/2743014.

14 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. doi:10.1016/0304-3975(94)90242-9.

15 Prateek Karandikar and Philippe Schnoebelen. Decidability in the logic of subsequences and supersequences. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015*, volume 45 of *LIPIcs*, pages 84–97, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.84.

16 Alexander Kartzow and Thomas Weidner. Model checking constraint LTL over trees, 2015. arXiv:1504.06105.

17 Dietrich Kuske. Theories of orders on the set of words. *RAIRO Theor. Informatics Appl.*, 40(1):53–74, 2006. doi:10.1051/ita:2005039.

18 Dietrich Kuske and Georg Zetzsche. Languages ordered by the subword order. In *Foundations of Software Science and Computation Structures – 22nd International Conference, FOSSACS 2019, Held as Part of ETAPS 2019, Proceedings*, volume 11425 of *Lecture Notes in Computer Science*, pages 348–364. Springer, 2019. doi:10.1007/978-3-030-17127-8_20.

19 Leonid Libkin, Wim Martens, and Domagoj Vrgoc. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, 2016. doi:10.1145/2850413.

20 Antonio Lozano and José L Balcázar. The complexity of graph problems for succinctly represented graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 277–286. Springer, 1989.

21 G. S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of The Ussr-sbornik*, 32:129–198, 1977.

22 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004. doi:10.1145/1013560.1013562.

23 Dominik Peteler and Karin Quaas. Deciding emptiness for constraint automata on strings with the prefix and suffix order. In *47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022*, volume 241 of *LIPIcs*, pages 76:1–76:15, 2022. `doi:10.4230/LIPIcs.MFCS.2022.76`.

24 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004. `doi:10.1145/990308.990312`.

25 Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

26 Luc Segoufin and Szymon Toruńczyk. Automata based verification over linearly ordered data domains. In *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011*, volume 9 of *LIPIcs*, pages 81–92. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. `doi:10.4230/LIPIcs.STACS.2011.81`.

The Best of Both Worlds: Model-Driven Engineering Meets Model-Based Testing

P. H. M. van Spaendonck[1]
Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands

Tim A. C. Willemse[1]
Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
ESI (TNO), Eindhoven, The Netherlands

Abstract

We study the connection between stable-failures refinement and the ioco conformance relation. Both behavioural relations underlie methodologies that have gained traction in industry: stable-failures refinement is used in several commercial Model-Driven Engineering tool suites, whereas the ioco conformance relation is used in Model-Based Testing tools. Refinement-based Model-Driven Engineering approaches promise to generate executable code from high-level models, thus guaranteeing that the code upholds specified behavioural contracts. Manual testing, however, is still required to gain confidence that the model-to-code transformation and the execution platform do not lead to unexpected contract violations. We identify conditions under which also this last step in the design methodology can be automated using the ioco conformance relation and the associated tools.

2012 ACM Subject Classification Theory of computation; Software and its engineering → Formal software verification

Keywords and phrases stable-failures refinement, Model-Driven Engineering, input output conformance, Input Output Labelled Transition Systems, internal choice

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.4

Funding *P. H. M. van Spaendonck*: This publication is part of the PVSR project (project number 17933) of the MasCot research programme which is financed by the Dutch Research Council (NWO).

Acknowledgements The authors would like to thank Rutger van Beusekom (Verum) for pointing them to the research question that led to this work. Furthermore, the authors would like to thank Jan Tretmans, and the anonymous referees, for feedback on an earlier version of this work.

1 Introduction

Formal Methods excel in eliminating subtle issues in complex software and system designs. Unfortunately, they are often perceived as complicated and inaccessible. For long, this sentiment has been a major reason for the slow industrial uptake of such methods. At the same time, Model-Driven Engineering (MDE), which promotes the use of Domain-Specific Languages (DSL) and code generation from models written in such languages, has managed to gain traction. MDE's success is in large part due to the close to perfect fit of a DSL and its application domain, which is in sharp contrast to the gap between generic Formal Methods and their domain of application.

[1] corresponding author

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 4; pp. 4:1–4:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Currently, we are witnessing the adoption of *formal* MDE approaches, in which the DSL is coupled to a design methodology that advocates a stepwise, compositional approach based on behavioural contracts (sometimes referred to as *service* or *behavioural interface* specification) of components. Commercial approaches of this kind are, e.g., Verum's Dezyne methodology [21, 20] and Cocotec's Coco platform [6].

Underlying the stepwise approach is typically a notion of refinement; for instance, the Dezyne methodology essentially utilises CSP's *stable-failures refinement* [15, 10]. The central idea is that the code that is generated from a model refines its behavioural contract, provided that the model refines the same contract. This way, the code for entire constellations of – guaranteed seemlessly cooperating – components can be generated with little effort.

One step often overlooked, however, is the fact that the model that is being verified is not identical to the code that is executed: even if the code generator is flawless, the behaviour of the component still depends on the execution platform, its operating system, the compilers used, *etcetera*. As a result, testing is still required to gain confidence in the correct execution of the generated code.

In practice, testing is still a largely manual and time-consuming activity; at best scripting is used to automatically execute a number of manually crafted test cases. Model-Based Testing (MBT) is a formal approach to testing that aims to improve on that situation. Tretmans' conformance theory [17, 18] is one of the most widely used testing theories, which has even found commercial use. As a starting point, MBT approaches take a formal specification, describing the system-under-test, and automatically derive tests from that specification, thus saving time on manually constructing and executing test cases, and maintaining these as the specification (and implementation) evolve.

To enable reasoning about implementations, formal approaches to testing typically assume that there is some (otherwise unknown) model with specific characteristics that underlies the actual implementation. This is sometimes referred to as the *testing assumption*. For instance, Tretmans [17, 18] assumes that implementations behave as input enabled Labelled Transition Systems with inputs and outputs. Weiglhofer and Wotawa [26] observe that this class of models is not quite suited in asynchronous settings and advocate *internal choice* Labelled Transition Systems with inputs and outputs. Such transition systems accept inputs only in states that are stable and no longer able to produce outputs. Crucially, implementations that are obtained through the MDE approach often fall in this class: these generally employ a *run-to-completion* semantics that assumes a component is ready for input only when it has finished processing the previous input.

Combining formal MDE approaches and MBT approaches seems natural and beneficial, but in practice, the two do not appear to match. Indeed, it is part of folklore that Tretmans' conformance theory, viz. ioco, is impossible to reconcile with theories of refinement such as the stable-failures refinement: as we also show in this paper, there are implementations that formally refine their specifications, but that nevertheless do not pass tests derived from such specifications. *Vice versa*, implementations that pass all tests derived from a given specification do not necessarily refine that specification.

At the same time, there are specifications and implementations for which stable-failures refinement and ioco both (do not) hold, suggesting there may be some room for combining the MBT and MDE methodologies in practice. We address this issue in this paper. More specifically, we study conditions under which Tretmans' ioco conformance relation can be used to assess the quality of implementations under the assumption (or guarantee) that the implementation is a stable-failures refinement of its specification. Our contributions are threefold:

- We characterise experiments that can be deduced from a specification – so-called *stable-failures testable traces* – for which ioco is guaranteed not to reject implementations that are a stable-failures refinement of that specification;
- We show that, surprisingly, for the class of internal choice Labelled Transition Systems of Weiglhofer and Wotawa [24, 26, 12, 13], the set of stable-failures testable traces coincides with Tretmans' suspension traces, implying that off-the-shelf ioco-based tooling can be used to test these implementations;
- We validate our theory in practice through a proof-of-concept implementation. In particular, we assess whether industrial-grade executable code, obtained using Verum's Dezyne methodology:
  - passes all tests automatically derived from its behavioural contract, and
  - fails tests when subtle mutations are introduced in the code.

Related Work. Several authors have attempted to equip the CSP theory with a testing theory. Cavalcanti and Gaudel [3] instantiate Gaudel's testing theory [7] for the (divergence-free fragment of the) CSP language and compare failures refinement to the *conf* relation. Their work, unlike ours, does not fundamentally distinguish inputs and outputs, contrary to, e.g., ioco. Sound and complete test suites for CSP's refinement relation are studied in [14]. In [4], and also later in [5], CSP is equipped with the notion of input and output. The authors use this distinction, in contrast to our work, to modify the stable-failures refinement to define a *new* refinement relation that is stronger than ioco on input enabled CSP processes. In [25], the authors give a denotational characterisation of an ioco-inspired conformance relation, in the context of a CSP-like process algebra. They show that, when applied to processes representing the suspension automata underlying a given specification and implementation, their relation coincides with Tretmans' ioco. Related to these approaches, in [11], the authors introduce a conformance relation called *CSP input-output conformance* to test systems that are both input and output enabled. They exploit use case templates to generate test cases by means of counterexamples to stable failures refinement. Finally, in [1], the authors coin *input-output tock-CSP refinement* and study its correspondence to a timed variant of ioco, called tioco [16], showing that the latter is weaker than their refinement relation.

In the broader scope, there have been several studies looking at the ioco relation from the perspective of refinement theories and game theory. For instance, in [23], the authors observe that ioco is non-compositional – in contrast to a proper refinement relation – prompting the authors to weaken the ioco relation. Their relation coincides with ioco when specifications have no under-specified inputs (for a more detailed discussion, we refer to, e.g. [19]). In [9], the authors compare ioco to alternating trace containment, a refinement relation in the setting of game theory and formal verification. They omit internal transitions (also known as *silent* steps) from their model, but their treatment does cover quiescence. The connection between testing theory and game theory had been previously studied by Van den Bos and Stoelinga [22].

Paper outline. Our paper is organised as follows. In Section 2, we introduce stable-failures refinement and Tretmans' ioco theory. Then, in Section 3 we introduce stable-failures testable traces and study their role in testing implementations that refine their specifications. In Section 4, we identify conditions that allow for proving stable-failures refinement using ioco. Section 5 we describe our experiments with the theory we developed, and we draw conclusions and sketch future work in Section 6.

2 Preliminaries

The behaviour of a system is typically formalised using (variations of) labelled transition systems (LTSs). Actions, taken from a sufficiently large alphabet Act, represent the observables of a system. We presuppose a constant $\tau \notin \mathsf{Act}$ to represent an unobservable action; the set Act_τ denotes the set $\mathsf{Act} \cup \{\tau\}$.

▶ **Definition 1.** *A labelled transition system (LTS) over* Act *is a tuple* $\langle S, \hat{s}, \rightarrow \rangle$*, where* S *is a set of states,* $\hat{s} \in S$ *is the initial state and* $\rightarrow \subseteq S \times \mathsf{Act}_\tau \times S$ *is the transition relation. We denote the set of LTSs over* Act *by* $\mathcal{LTS}(\mathsf{Act})$*.*

We often refer to a given LTS $\langle S, \hat{s}, \rightarrow \rangle$ by its initial state $\hat{s}$. We write $s \xrightarrow{x} s'$ rather than $(s, x, s') \in \rightarrow$; moreover, we write $s \xrightarrow{x}$ when $s \xrightarrow{x} s'$ for some s', and $s \not\xrightarrow{x}$ when $s \xrightarrow{x}$ does not hold. The transition relation is lifted to a relation over $S \times \mathsf{Act}_\tau^* \times S$ in the usual manner, and we lift the notation introduced for $\rightarrow$ accordingly. We say that a word $w \in \mathsf{Act}_\tau^*$ is a *concrete trace* of an LTS $\hat{s}$ iff $\hat{s} \xrightarrow{w}$, and we say that a state s is *reachable* exactly when $\hat{s} \xrightarrow{w} s$ for some concrete trace w.

A further generalisation of $\rightarrow$ to a relation over words of observable actions $\Longrightarrow \subseteq S \times \mathsf{Act}^* \times S$ is obtained as the smallest relation satisfying the following rules:

$$\frac{}{s \overset{\epsilon}{\Longrightarrow} s} \qquad \frac{s \overset{w}{\Longrightarrow} s'' \quad s'' \xrightarrow{x} s' \quad x \neq \tau}{s \overset{w\,x}{\Longrightarrow} s'} \qquad \frac{s \overset{w}{\Longrightarrow} s'' \quad s'' \xrightarrow{\tau} s'}{s \overset{w}{\Longrightarrow} s'}$$

We adopt the notational conventions we introduced earlier for $\rightarrow$ also for $\Longrightarrow$. The set of *traces* of a states s is denoted $\mathsf{Traces}(s) = \{w \in \mathsf{Act}^* \mid s \overset{w}{\Longrightarrow}\}$. For a set of states S', we define $\mathsf{Traces}(S') = \bigcup_{s' \in S'} \mathsf{Traces}(s')$.

▶ **Definition 2.** *Let* $\langle S, \hat{s}, \rightarrow \rangle$ *be an LTS. For arbitrary state* $s \in S$ *and set of states* $S' \subseteq S$*, we define:*

1. $\mathsf{init}(s) = \{x \in \mathsf{Act}_\tau \mid s \xrightarrow{x}\}$ *and* $\mathsf{init}(S') = \bigcup_{s' \in S'} \mathsf{init}(s')$*;*
2. $\mathsf{Sinit}(s) = \{x \in \mathsf{Act} \mid s \overset{x}{\Longrightarrow}\}$ *and* $\mathsf{Sinit}(S') = \bigcup_{s' \in S'} \mathsf{Sinit}(s')$*;*
3. $\mathsf{stable}(s)$ *iff* $\tau \notin \mathsf{init}(s)$*, and* $\mathsf{stable}(S')$ *iff for all* $s' \in S$ *we have* $\mathsf{stable}(s')$*.*

We say that an LTS $\langle S, \hat{s}, \rightarrow \rangle$ is *convergent* when none of its states $s \in S$ are divergent, i.e., no state in S is the start of an infinite sequence of τ-steps.

A set of observable actions $X \subseteq \mathsf{Act}$ is a *refusal* for a state s exactly when $\mathsf{init}(s) \cap X = \emptyset$. Given a state s, we say that the pair (w, X) is a *failure* for state s when there is some s' such that $\mathsf{stable}(s')$, $s \overset{w}{\Longrightarrow} s'$ and $\mathsf{init}(s') \cap X = \emptyset$. The set of failures of a state s is denoted $\mathsf{Failures}(s)$, and defined formally as follows:

$$\mathsf{Failures}(s) = \{(w, X) \in \mathsf{Act}^* \times 2^{\mathsf{Act}} \mid \exists s' : s \overset{w}{\Longrightarrow} s' \wedge \mathsf{stable}(s') \wedge X \cap \mathsf{init}(s') = \emptyset\}$$

We next recall a classical notion of refinement underlying process algebras such as CSP, see, e.g. [15, 10].

▶ **Definition 3.** *Let* $\langle S, \hat{s}, \rightarrow \rangle$ *be an LTS. For states* $s, t \in S$*, we define* $s \sqsubseteq_F t$ *iff* $\mathsf{Traces}(t) \subseteq \mathsf{Traces}(s)$ *and* $\mathsf{Failures}(t) \subseteq \mathsf{Failures}(s)$*. We say* t *is a* stable-failures refinement *of* s *iff* $s \sqsubseteq_F t$*.*

When interacting with an actual implementation, the initiative to communicate is often not symmetric: the implementation can receive stimuli from its environment and produce events that are to be consumed by the environment. We therefore refine the LTS model to incorporate a distinction between *inputs* and *outputs*.

▶ **Definition 4.** *An input-output labelled transition system over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *is an LTS* $\langle S, \hat{s}, \rightarrow \rangle$ *over* Act *in which* Act *is partitioned into a set* Act_I *of inputs and a set* Act_U *of outputs. We denote the set of input-output labelled transition systems (IOLTS) over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *by* $\mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$*.*

As a notational convention we distinguish inputs from outputs by adding question- (?) and exclamation-mark (!) symbols, respectively, in our examples. We stress that these decorations are *not* part of action names. States are *quiescent* when they are stable and refuse to produce output. Quiescence, defined formally below, is a crucial element in many testing theories, needed to disqualify implementations that fail to produce output when not expected.

▶ **Definition 5.** *Let* $\langle S, \hat{s}, \rightarrow \rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*, and let* $s \in S$*. We say that* s *is* quiescent*, denoted* $\delta(s)$*, iff* $\mathsf{stable}(s)$ *and* $\mathsf{init}(s) \cap \mathsf{Act}_U = \emptyset$*.*

We say that an IOLTS $\langle S, \hat{s}, \rightarrow \rangle$ is an *internal choice IOLTS* iff inputs are only specified in quiescent states; i.e., exactly when for all $s \in S$ for which $\mathsf{init}(s) \cap \mathsf{Act}_I \neq \emptyset$, also $\delta(s)$ holds true. We denote the set of internal choice IOLTSs over $(\mathsf{Act}_I, \mathsf{Act}_U)$ by $\mathcal{IOLTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$.

Quiescence is typically treated as an output of the system, i.e., an observable of an implementation under test. Let $\delta \notin \mathsf{Act}$ be a special constant denoting the observation of quiescence, and let Act_δ denote the set $\mathsf{Act} \cup \{\delta\}$.

▶ **Definition 6.** *Let* $\langle S, \hat{s}, \rightarrow \rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*, and let* $s \in S$*.*

- *The* outputs *enabled in* s*, denoted* $\mathsf{out}(s)$*, is defined as* $\mathsf{out}(s) = \{\delta \mid \delta(s)\} \cup (\mathsf{Act}_U \cap \mathsf{init}(s))$*;*
- *The* inputs *enabled in* s*, denoted* $\mathsf{in}(s)$*, is defined as* $\mathsf{in}(s) = \mathsf{Act}_I \cap \mathsf{Sinit}(s)$*.*

For a set of states $S' \subseteq S$*, we define* $\mathsf{out}(S') = \bigcup_{s' \in S'} \mathsf{out}(s')$ *and* $\mathsf{in}(S') = \bigcap_{s' \in S'} \mathsf{in}(s')$*.*

The notion of a *suspension trace* incorporates the observation of quiescence also in our observations of the behaviour of an implementation over time.

▶ **Definition 7.** *Let* $\langle S, \hat{s}, \rightarrow \rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*, and let* $s \in S$*. We say that a sequence of events* $w \in \mathsf{Act}_\delta^*$ *is a* suspension trace *of* s *iff* $w \in \mathsf{Traces}(s_\Delta)$ *in the IOLTS* $\Delta(\hat{s})$ *over* $(\mathsf{Act}_I, \mathsf{Act}_U \cup \{\delta\})$*, where* $\Delta(\hat{s}) = \langle S_\Delta, \hat{s}_\Delta, \rightarrow_\Delta \rangle$ *is defined as follows:*

- $S_\Delta = \{s'_\Delta \mid s' \in S\}$*;*
- $\rightarrow_\Delta = \{(s'_\Delta, x, s''_\Delta) \mid s' \xrightarrow{x} s''\} \cup \{(s', \delta, s') \mid \delta(s')\}$*.*

The set of suspension traces of a state $s \in S$ *is denoted* $\mathsf{STraces}(s)$*.*

We generalise the relation $\rightarrow_\Delta$ to $\Longrightarrow_\Delta$ as before and we allow ourselves to write $s \overset{w}{\Longrightarrow}_\Delta s'$, for states s, s' of an IOLTS $\langle S, \hat{s}, \rightarrow \rangle$, when we in fact mean $s_\Delta \overset{w}{\Longrightarrow}_\Delta s'_\Delta$.

▶ **Definition 8.** *Let* $\langle S, \hat{s}, \rightarrow \rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*. For states* $s \in S$ *and suspension traces* $w \in \mathsf{STraces}(s)$*, we define* s after $w = \{s' \in S \mid s \overset{w}{\Longrightarrow}_\Delta s'\}$*. For sets of states* $S' \subseteq S$ *we define* S' after $w = \bigcup_{s' \in S'} s'$ after w*.*

Formal testing theories usually build upon the assumption that an implementation can be captured adequately in a submodel of IOLTSs. We recall two such submodels, viz., the *input output transition systems*, used in Tretmans' testing theory [17, 18] and the *internal choice input output transition systems*, introduced by Weiglhofer and Wotawa [24, 26].

Tretmans' input-output transition systems are IOLTSs with the additional assumption that inputs will always be accepted. That is, implementations are assumed to be *input enabled*.

▶ **Definition 9.** *Let* $\langle S, \hat{s}, \rightarrow, s_0\rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*. A state* $s \in S$ *is* input-enabled *iff* $\mathsf{Act}_I \subseteq \mathsf{Sinit}(s)$*. The IOLTS* $\hat{s}$ *is an* input output transition system *(IOTS) iff every state* $s \in S$ *is input-enabled. We denote the class of input output transition systems ranging over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *by* $\mathcal{IOTS}(\mathsf{Act}_I, \mathsf{Act}_U)$*.*

Weiglhofer and Wotawa's model of internal choice input output transition systems relax the requirement that implementations must be input-enabled at all times. Instead, they require that only quiescent states are input-enabled, and inputs are only accepted in quiescent states. Their model better fits with implementations that rely on some form of *run to completion*.

▶ **Definition 10** (Internal choice IOTS)**.** *An IOLTS* $\langle S, \hat{s}, \rightarrow\rangle$ *is an* internal choice input output transition system *over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *if for all states* $s \in S$*:*

1. *if* $\delta(s)$*, then* $\mathsf{Act}_I \subseteq \mathsf{init}(s)$
2. *if* $\mathsf{init}(s) \cap \mathsf{Act}_I \neq \emptyset$ *then* $\delta(s)$*.*

We denote the class of internal choice input output transition systems over $(\mathsf{Act}_I, \mathsf{Act}_U)$ *by* $\mathcal{IOTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$*.*

Testing is used to assess whether a given implementation conforms to its specification. Several conformance relations have been proposed in the literature, and one of the most prominent ones is *input output conformance* by Tretmans [17, 18]. This conformance relation formalises when an implementation, assumed to behave as an input output transition system, complies to a given specification. Following e.g. [9], we assume here that implementations can behave, more generally, as input output labelled transition systems.

▶ **Definition 11.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$ *be (a model of) an implementation and specification, respectively. We say that* imp input output conforms *to* spec*, denoted* $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$*, iff for all* $w \in \mathsf{STraces}(\mathsf{spec})$ *we have:*

1. $\mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w) \subseteq \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$*,*
2. $\mathsf{in}(\mathsf{imp}\ \mathsf{after}\ w) \supseteq \mathsf{in}(\mathsf{spec}\ \mathsf{after}\ w)$*.*

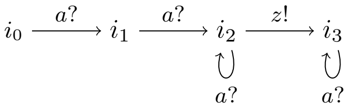
We remark that condition 2, on the inputs, can be dropped in the above definition in case the implementation is input enabled, thus simplifying to the definition that can be found in [17, 18]. After all, input enabledness guarantees that inputs can always be consumed by the implementation.

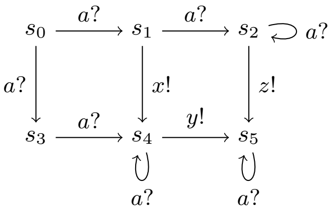
3 Testing Refinements of Specifications

Refinement relations are particularly useful in a design methodology in which a system is successively refined into smaller components, where, at each step, the relevant artefacts can be related by a stable-failures refinement. Once the models for (sub)components are sufficiently detailed and simple, implementing these as executable code should be reasonably straightforward and is even done automatically in formal MDE approaches.

Despite the simplicity and details of these models, the conversion to executable code may introduce bugs. Even if no bugs are introduced in this step, the platform on which the code runs may inject issues not foreseen at the time of the design. Conformance testing is therefore a step that cannot be omitted, but as the following example illustrates, the ioco-conformance relation may flag implementations to be incorrect, despite these being correct with respect to stable-failures refinement.

▶ **Example 12.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).





Observe that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ holds true. However, imp ioco spec does not hold, since $\mathsf{out}(s_0 \text{ } \mathsf{after} \text{ } a\,\delta\,a) = \{y\}$, whereas $\mathsf{out}(i_0 \text{ } \mathsf{after} \text{ } a\,\delta\,a) = \{z\}$. ⌟

Conceptually, the non-conformance in the above example is caused by the ability to continue testing beyond observations of quiescence. This suggests that, in general, we cannot safely test the full specification for all its suspension traces. The question thus arises what subset of the behaviour, modelled by a specification, is available to us for testing. We coin a set of suspension traces for which we subsequently argue that testing for these cannot lead to verdicts that conflict with previously established refinements.

▶ **Definition 13.** *Let* $\mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$ *be an arbitrary IOLTS. A suspension trace* w *of* spec *is* stable-failures testable *exactly when for all prefixes* $v\,\delta\,x$ *of* w*, with* $x \in \mathsf{Act}_\delta$*, we have* spec after $v\,x = \mathsf{spec}$ after $v\,\delta\,x$*. The set of all stable-failures testable suspension traces is denoted* $\mathsf{TTraces}(\mathsf{spec})$*.*

One may remark that the set Act_δ that x may range over in the above definition is too liberal. Indeed, since suspension traces are anomaly-free [27], x cannot be an output. Restricting the set of symbols that x ranges over to $\mathsf{Act}_I \cup \{\delta\}$ would therefore yield an equivalent, though in practice somewhat more cumbersome, definition.

We start by noting two relevant properties of the set of stable-failures testable traces of a specification.

▶ **Lemma 14.** *We have* $\mathsf{Traces}(\mathsf{spec}) \subseteq \mathsf{TTraces}(\mathsf{spec})$*.*

Proof. Pick some arbitrary $w \in \mathsf{Traces}(s)$. Since $\mathsf{Traces}(s) \subseteq \mathsf{STraces}(s)$, also $w \in \mathsf{Traces}(s)$. Moreover, since $w \in \mathsf{Traces}(s)$, also $w \in \mathsf{Act}^*$. Since there is no prefix of the shape $v\,\delta\,x$ in $w \in \mathsf{Act}^*$, we find that w is stable-failures testable. Hence $w \in \mathsf{TTraces}(\mathsf{spec})$. ◀

▶ **Lemma 15.** *The set* $\mathsf{TTraces}(\mathsf{spec})$ *is prefix closed.*

Proof. Pick some $w \in \mathsf{TTraces}(\mathsf{spec})$, and let w' be a prefix of w. Consider an arbitrary prefix $v\,\delta\,x$ of w'. Then $v\,\delta\,x$ is also a prefix of w. Since $w \in \mathsf{TTraces}(\mathsf{spec})$ we therefore have spec after $v\,x = \mathsf{spec}$ after $v\,\delta\,x$. But then also $w' \in \mathsf{TTraces}(\mathsf{spec})$. ◀

We next introduce the operators $\bar{\bar{w}}$ and $\bar{w}$ on suspension traces. In essence, these operators remove all δ-symbols (respectively, all but a terminal δ-symbol, if present) from a suspension trace.

▶ **Definition 16.** *Let* $x \in \mathsf{Act}_\delta$*,* $y \in \mathsf{Act}$*,* $v \in \mathsf{Act}^*$ *and* $w \in \mathsf{Act}_\delta^+$*. We define the operators* $\overline{\overline{\_}} : \mathsf{Act}_\delta^* \to \mathsf{Act}_\delta^*$ *and* $\overline{\_} : \mathsf{Act}_\delta^* \to \mathsf{Act}^*$ *as follows:*

$$\bar{\bar{\epsilon}} = \epsilon, \qquad \overline{\overline{y\,v}} = y\,\bar{\bar{v}}, \qquad \overline{\overline{\delta\,v}} = \bar{\bar{v}}$$
$$\bar{\epsilon} = \epsilon, \qquad \bar{x} = x, \qquad \overline{y\,w} = y\,\bar{w}, \qquad \overline{\delta\,w} = \bar{w}$$

Observe that in case $w \in \mathsf{Act}^*$, we have $\overline{w} = \overline{\overline{w}} = w$. In case $w \in \mathsf{Act}_\delta^*\mathsf{Act}^+$, we have $\overline{w} = \overline{\overline{w}}$, and in case $w \in \mathsf{Act}_\delta^*\delta^+$ we have $\overline{w} = \overline{\overline{w}}\,\delta$.

▶ **Lemma 17.** *Let* $\mathsf{spec} = \langle S, \hat{s}, \rightarrow\rangle$ *be an arbitrary IOLTS. For all* $w \in \mathsf{TTraces(spec)}$*,* $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{w}$*.*

Proof. The proof proceeds by means of an induction on the number of δ's appearing in w.

- Base case: w contains no δ-symbols. Then $w \in \mathsf{Traces(spec)}$ and since $\overline{w} = w$ for traces, we immediately find the desired $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{w}$.
- Induction: suppose that for all $z \in \mathsf{TTraces(spec)}$, containing n δ-symbols, we have $\mathsf{spec\ after}\ z = \mathsf{spec\ after}\ \overline{z}$. Pick some $w \in \mathsf{TTraces(spec)}$ containing $n+1$ δ-symbols. Then w must be of the shape $v\,\delta\,u$, with $u \in \mathsf{Act}^*$, and v containing n δ-symbols. We distinguish two cases:
  - Case $u = \epsilon$. Then $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ v\,\delta = (\mathsf{spec\ after}\ v)\ \mathsf{after}\ \delta$ By induction, the latter is equal to $(\mathsf{spec\ after}\ \overline{v})\ \mathsf{after}\ \delta$, which is equivalent to $\mathsf{spec\ after}\ \overline{v}\,\delta$. We distinguish two further cases:
    - * Case $\overline{v} \in \mathsf{Traces(spec)}$. Then $\overline{v}\,\delta = \overline{v\,\delta} = \overline{w}$, and consequently, $\mathsf{spec\ after}\ \overline{v}\,\delta = \mathsf{spec\ after}\ \overline{w}$.
    - * Case $\overline{v} \notin \mathsf{Traces(spec)}$. Then $\overline{v} = v'\,\delta$ for some $v' \in \mathsf{Traces(spec)}$ and therefore $\overline{v}\,\delta = v'\,\delta\,\delta$. Observe that we have $\mathsf{spec\ after}\ v'\,\delta\,\delta = \mathsf{spec\ after}\ v'\,\delta = \mathsf{spec\ after}\ \overline{v\,\delta} = \mathsf{spec\ after}\ \overline{w}$.

    In both cases, we are done.
  - Case $u \neq \epsilon$. We necessarily have $u = x\,u'$ for some x and u'. Then, by Definition 13, we have $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ v\,\delta\,x\,u' = \mathsf{spec\ after}\ v\,x\,u'$. Since $v\,x\,u'$ contains exactly n δ-symbols, we may conclude, by induction that $\mathsf{spec\ after}\ v\,x\,u' = \mathsf{spec\ after}\ \overline{v\,x\,u'}$. But $\overline{v\,x\,u'} = \overline{w}$, so we may conclude $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{w}$. ◀

▶ **Definition 18.** *We say that an IOLTS* spec *is stable-failures testable exactly when it satisfies* $\mathsf{STraces(spec)} = \mathsf{TTraces(spec)}$*.*

It may be clear that not every IOLTS is stable-failures testable. For instance, the specification depicted in Example 12 contains suspension traces that are not stable-failures testable: the sequence $a\,\delta\,a$, which we used to illustrate the non-conformance of the implementation to the specification is not stable-failures testable, since $s_0\ \mathsf{after}\ a\,\delta\,a = \{s_4\} \neq \{s_2, s_4\} = s_0\ \mathsf{after}\ a\,a$. On the other hand, the implementation depicted in the same example is stable-failures testable. The class of internal choice IOLTSs also turns out to be stable-failures testable, as asserted by the theorem below.

▶ **Theorem 19.** *Every internal choice IOLTS is stable-failures testable.*

Proof. Clearly, $\mathsf{TTraces(spec)} \subseteq \mathsf{STraces(spec)}$, so it suffices to prove $\mathsf{STraces(spec)} \subseteq \mathsf{TTraces(spec)}$. This can be shown using an induction on the length of the suspension traces.

- Base case $w = \epsilon$. Since $\epsilon \in \mathsf{Traces(spec)} \subseteq \mathsf{TTraces(spec)}$, we are done.
- Suppose that for $w \in \mathsf{STraces(spec)}$ of length n, we have $w \in \mathsf{TTraces(spec)}$. Let $x \in \mathsf{Act}_\delta$ be such that $w\,x \in \mathsf{STraces(spec)}$. Let $v\,\delta\,y$ be a prefix of $w\,x$. If $v\,\delta\,y$ is a prefix of w, then we may conclude $\mathsf{spec\ after}\ v\,y = \mathsf{spec\ after}\ v\,\delta\,y$ from our induction hypothesis and we are done.

  So suppose that $v\,\delta\,y = w\,x$. It now suffices to prove that $\mathsf{spec\ after}\ v\,y = \mathsf{spec\ after}\ v\,\delta\,y$. Note that $\mathsf{spec\ after}\ v\,y \supseteq \mathsf{spec\ after}\ v\,\delta\,y$ follows from the fact that observations of δ do not change state, so it suffices to prove $\mathsf{spec\ after}\ v\,y \subseteq \mathsf{spec\ after}\ v\,\delta\,y$. Pick some $s \in \mathsf{spec\ after}\ v\,y$. From $w\,x = v\,\delta\,y \in \mathsf{STraces(spec)}$ we may conclude that $y \notin \mathsf{Act}_U$. We distinguish two cases:

- Case $y = \delta$. Then it immediately follows that also $s \in \mathsf{spec\ after}\ v\,\delta\,y$ and we are done.
- Case $y \neq \delta$. This implies that $y \in \mathsf{Act}_I$. Since spec is an internal choice IOLTS, we find that there must be some $s' \in \mathsf{spec\ after}\ v$ such that $\delta(s')$ and $s' \overset{y}{\Longrightarrow}_{\Delta} s$. Let s' be such. Since $s' \overset{\delta}{\Longrightarrow}_{\Delta} s'$, we may conclude that also $s \in \mathsf{spec\ after}\ v\,\delta\,y$. ◀

We next formally relate the failures refinement theory to the input output conformance testing theory. Lemma 20 states that the outputs of implementations that are a stable-failures refinement of a given specification can be safely tested using stable-failures testable suspension traces. Likewise, Lemma 21, states that the inputs of convergent implementations that are a stable-failures refinement of a given specification can be safely tested using stable-failures testable suspension traces.

▶ **Lemma 20.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$*. Assume that* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *holds true. Then* $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$ *for all* $w \in \mathsf{TTraces}(\mathsf{spec})$*.*

Proof. Suppose that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. Towards a contradiction, assume that for some $w \in \mathsf{TTraces}(\mathsf{spec})$ we do not have $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$. Without loss of generality, assume that w is the shortest such trace. This implies, in particular, that w is not of the form $v\,\delta$, since such a suspension trace cannot give rise to the desired contradiction, and therefore $\overline{w} \in \mathsf{Traces}(\mathsf{spec})$. Note that we also can conclude that $\mathsf{out}(\mathsf{imp\ after}\ w) \neq \emptyset$ and hence $w \in \mathsf{STraces}(\mathsf{imp})$. Since imp is quiescence-reducible [27], we therefore also have $\overline{w} \in \mathsf{Traces}(\mathsf{imp})$. By definition, $\mathsf{imp\ after}\ w \subseteq \mathsf{imp\ after}\ \overline{w}$. Consequently, $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{imp\ after}\ \overline{w})$. Furthermore, using Lemma 17 we may conclude that $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{w}$, so also $\mathsf{out}(\mathsf{spec\ after}\ w) = \mathsf{out}(\mathsf{spec\ after}\ \overline{w})$.

Let $X = \mathsf{out}(\mathsf{imp\ after}\ \overline{w}) \setminus \mathsf{out}(\mathsf{spec\ after}\ \overline{w})$. We distinguish two cases:

- Case $\delta \in X$. Then, $(\overline{w}, \mathsf{Act}_U) \in \mathsf{Failures}(\mathsf{imp})$, but $(\overline{w}, \mathsf{Act}_U) \notin \mathsf{Failures}(\mathsf{spec})$. Since $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$, this cannot be the case. Contradiction.
- Case $\delta \notin X$. Pick $x \in X$. Then $\overline{w}\,x \in \mathsf{Traces}(\mathsf{imp})$, but $\overline{w}\,x \notin \mathsf{Traces}(\mathsf{spec})$. Again, since $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$, this cannot be the case. Contradiction.

Since both cases lead to a contradiction, we may conclude that for all $w \in \mathsf{TTraces}(\mathsf{spec})$ we have $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$. ◀

▶ **Lemma 21.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$*. Assume* imp *is convergent and assume* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *holds true. Then* $\mathsf{in}(\mathsf{spec\ after}\ w) \subseteq \mathsf{in}(\mathsf{imp\ after}\ w)$ *for all* $w \in \mathsf{TTraces}(\mathsf{spec})$*.*

Proof. Assume that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. Suppose that for $w \in \mathsf{TTraces}(\mathsf{spec})$, $\mathsf{in}(\mathsf{spec\ after}\ w) \subseteq \mathsf{in}(\mathsf{imp\ after}\ w)$ does not hold. Note that this implies that $\mathsf{in}(\mathsf{spec\ after}\ w) \neq \emptyset$. Pick such w and some input $a \in \mathsf{in}(\mathsf{spec\ after}\ w) \setminus \mathsf{in}(\mathsf{imp\ after}\ w)$. By definition, this means that for all $s \in \mathsf{spec\ after}\ w$ we have $s \overset{a}{\Longrightarrow}$. By Lemma 17, $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{w}$, so also $s \overset{a}{\Longrightarrow}$ for all $s \in \mathsf{spec\ after}\ \overline{w}$. Observe that this also implies that for all stable states $t \in \mathsf{spec\ after}\ \overline{w}$, if any, we have $t \xrightarrow{a}$. We distinguish two cases:

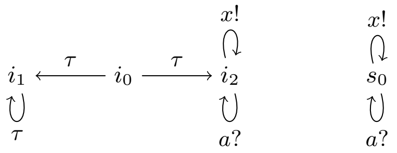
- $\overline{w} \notin \mathsf{Traces}(\mathsf{spec})$. Then $\overline{w} = \overline{\overline{w}}\,\delta$ and since $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{\overline{w}}\,\delta \neq \emptyset$, there is some $t \in \mathsf{spec\ after}\ \overline{\overline{w}}\,\delta$ satisfying $\delta(t)$, and which is therefore stable. Since for every stable state $t \in \mathsf{spec\ after}\ \overline{\overline{w}}\,\delta$ we have $t \in \mathsf{spec\ after}\ \overline{\overline{w}}$, we may conclude that $(\overline{\overline{w}}, \{a\}) \notin \mathsf{Failures}(\mathsf{spec})$.
- $\overline{w} \in \mathsf{Traces}(\mathsf{spec})$. Since in that case $\overline{w} = \overline{\overline{w}}$, we again conclude that $(\overline{\overline{w}}, \{a\}) \notin \mathsf{Failures}(\mathsf{spec})$.

From the above, we thus conclude that $(\overline{\overline{w}}, \{a\}) \notin \mathsf{Failures}(\mathsf{spec})$. We will next argue that $(\overline{\overline{w}}, \{a\}) \in \mathsf{Failures}(\mathsf{imp})$. Since this contradicts $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$, we may conclude that $\mathsf{in}(\mathsf{spec\ after}\ w) \subseteq \mathsf{in}(\mathsf{imp\ after}\ w)$, finishing the proof.

Concerning the remaining proof obligation $(\overline{\overline{w}}, \{a\}) \in \mathsf{Failures}(\mathsf{imp})$, we reason as follows. Since $a \notin \mathsf{in}(\mathsf{imp}\ \mathsf{after}\ w)$ and imp is convergent, we conclude that there must be some state $s \in \mathsf{imp}\ \mathsf{after}\ w$ such that $\mathsf{stable}(s)$ and $s \stackrel{a}{\not\rightarrow}$. Let s be such a state. By definition, we have $\mathsf{imp}\ \mathsf{after}\ w \subseteq \mathsf{imp}\ \mathsf{after}\ \overline{\overline{w}}$, so also $s \in \mathsf{imp}\ \mathsf{after}\ \overline{\overline{w}}$. But then $(\overline{\overline{w}}, \{a\}) \in \mathsf{Failures}(\mathsf{imp})$. ◀

One might wonder whether the convergence condition is strictly needed. The example below illustrates that this condition can indeed not be dropped in general.

▶ **Example 22.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).



Observe that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ holds true. Moreover, note that due to the τ-loop, imp is not convergent. By Lemma 20, we find that for every $w \in \mathsf{STraces}(\mathsf{spec})$, we have $\mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w) \subseteq \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$; this is readily checked. However, we have $\mathsf{in}(\mathsf{spec}\ \mathsf{after}\ \epsilon) = \{a\} \neq \emptyset = \mathsf{in}(\mathsf{imp}\ \mathsf{after}\ \epsilon)$. Consequently, $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$ does not hold true. ⌟

The theorem below follows immediately from the two lemmata above.

▶ **Theorem 23.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$*. Assume* imp *is convergent. If* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *then also for all* $w \in \mathsf{TTraces}(\mathsf{spec})$*, we have:*

1. $\mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w) \subseteq \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$*, and*
2. $\mathsf{in}(\mathsf{imp}\ \mathsf{after}\ w) \supseteq \mathsf{in}(\mathsf{spec}\ \mathsf{after}\ w)$*.*

Theorem 23 specialises to standard ioco in case the specification is an internal choice IOLTS and the implementation is convergent, as claimed by the corollary below.

▶ **Corollary 24.** *Let* $\mathsf{spec} \in \mathcal{IOLTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$ *and* $\mathsf{imp} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$*. Suppose* imp *is convergent. Then* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *implies* $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$*.*

We finish with the observation that in case the specification is an internal choice IOLTS and the implementation is an internal choice IOTS, the requirement on the implementation being convergent can be dropped, see the corollary below.

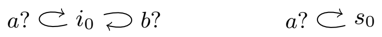
▶ **Corollary 25.** *Let* $\mathsf{spec} \in \mathcal{IOLTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$ *and* $\mathsf{imp} \in \mathcal{IOTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$*. Then* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *implies* $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$*.*

4 Stable Failures Refinement through Testing

We next identify conditions under which we may conclude that a model of an implementation is a stable-failures refinement of a given specification after exhaustively testing a faithful implementation of that model.

Let us first observe that if the specification that is used for testing is not input enabled, we will not be able to establish a stable-failures refinement relation between the specification and the implementation. Since the ioco-conformance relation allows for partial specifications, only those parts that are specified are tested for, and other parts are ignored, resulting in potentially labelling such an implementation as one that conforms to its specification. As a result, inputs that are not specified cannot be excluded to be part of some conforming implementation and will thus lead to trace inclusion violations. This is illustrated by the following (trivial) example.

▶ **Example 26.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).



Clearly, we have imp ioco spec, but the trace $b \in \mathsf{Traces}(i_0)$ is not present in $\mathsf{Traces}(s_0)$, thus contradicting spec $\sqsubseteq_F$ imp. ⌟

Consequently we can only assess that an implementation refines a specification if the latter is "at least as input enabled" as the implementation that we are (black box) testing for. In Tretmans original testing theory, but also in Weiglhofer and Wotawa's theory, the input enabledness of the implementation is typically part of the testing assumption, which, depending on the applications at hand, state that the implementation is either always input enabled (IOTSs), or input enabled exactly (and only) in quiescent states (internal choice IOTSs). We therefore confine our analysis to implementations that can be modelled as an IOTS or an internal choice IOTS, and we study specifications that – in terms of their input enabledness – fit these assumptions. For these systems, we have the following observation:

▶ **Lemma 27.** *Let* spec, imp *be IOLTSs. Suppose that either:*

- *both* spec *and* imp *are IOTSs, or*
- *both* spec *and* imp *are internal choice IOTSs.*

Then imp ioco spec *implies* $\mathsf{Traces}(\mathsf{imp}) \subseteq \mathsf{Traces}(\mathsf{spec})$.

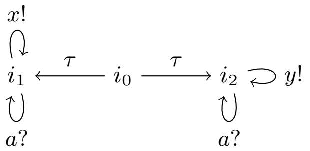
Proof. Suppose that imp ioco spec. Let $w \in \mathsf{Traces}(\mathsf{imp})$ be such that $w \notin \mathsf{Traces}(\mathsf{spec})$, and, without loss of generality, assume that there is no shorter trace. Observe that $w \neq \epsilon$, since ϵ is a weak trace of both imp and spec. Hence, w must be of the shape $v\,x$, for some trace $v \in \mathsf{Traces}(\mathsf{imp}) \cap \mathsf{Traces}(\mathsf{spec})$ and action $x \in \mathsf{Act}$. Let v and x be such.

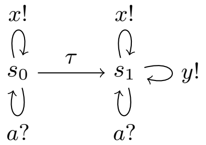
We first argue that $x \notin \mathsf{Act}_I$. Observe that this follows trivially in case imp and spec are both IOTSs, since spec would be required to accept input a at any moment. In case spec is an internal choice IOTS, we reason as follows. Towards a contradiction, assume that $x \in \mathsf{Act}_I$. Then $v\,x \notin \mathsf{Traces}(\mathsf{spec})$ can only be the case when $\delta \notin \mathsf{out}(\mathsf{spec\ after\ } v)$, since spec is input enabled only (and exactly) in quiescent states. Since $v\,x \in \mathsf{Traces}(\mathsf{imp})$, we must conclude that $\delta \in \mathsf{out}(\mathsf{imp\ after\ } v)$. But this violates our assumption that imp ioco spec. Hence, also in case imp and spec are internal choice IOTSs, we have $x \notin \mathsf{Act}_I$.

Consequently, $x \in \mathsf{Act}_U$ and therefore $x \in \mathsf{out}(\mathsf{imp\ after\ } v)$. Since $v \in \mathsf{STraces}(\mathsf{spec})$ and imp ioco spec, we also find $x \in \mathsf{out}(\mathsf{spec\ after\ } v)$. This implies that $v\,x \in \mathsf{Traces}(\mathsf{spec})$. Contradiction. Hence, $\mathsf{Traces}(\mathsf{imp}) \subseteq \mathsf{Traces}(\mathsf{spec})$. ◀

In view of the above result, assuming some form of input enabledness of the specification is essential for guaranteeing trace inclusion, which is an essential part of the refinement relation. However, input enabledness does little to establish the other essential part of the refinement relation, viz., the inclusion of the set of failures. This has to do with the fact that refinement allows for observing the refusals of individual actions, contrary to the ioco conformance relation. The next example illustrates the issue. We remark that the example uses an implementation that behaves as an IOTS, but this can be modified easily to show the same issue in internal choice IOTSs.

▶ **Example 28.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).





Note that imp ioco spec; in particular, $\mathsf{out}(i_0\ \mathsf{after}\ \epsilon) = \mathsf{out}(s_0\ \mathsf{after}\ \epsilon)$. Clearly, $(\epsilon, \{y\}) \notin \mathsf{Failures}(s_0)$ since stable state s_1 does not refuse y; of course, state s_0 does not offer action y, but since s_0 is unstable, its refusals are not taken into account. However, since i_1 is stable, $(\epsilon, \{y\}) \in \mathsf{Failures}(i_0)$. Therefore, $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ does not hold true. ⌟

The above example illustrates that, from the point of view of stable-failures refinement, output actions should be preserved and ultimately *determined*: τ-paths should eventually lead to states in which only "trivial output choices" can be made.

▶ **Definition 29.** *Let $\langle S, \hat{s}, \rightarrow\rangle$ be an IOLTS. We say that $\hat{s}$ is* ultimately determined *iff for all states $s \in S$ and all $x \in \mathsf{out}(s\ \mathsf{after}\ \epsilon)$ there is some $t \in s\ \mathsf{after}\ \epsilon$ such that $\mathsf{out}(t) = \{x\}$.*

Observe that the specification of Example 28 is not ultimately determined, since, e.g., there is no state $s \in s_0\ \mathsf{after}\ \epsilon$ such that $\mathsf{out}(s) = \{y\}$.

▶ **Proposition 30.** *For any IOTS* imp *and convergent, ultimately determined IOTS* spec *satisfying* imp ioco spec *we have* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$.

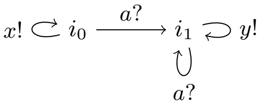
Proof. Suppose that imp ioco spec holds true for IOTSs imp and spec, and that spec is both convergent and ultimately determined. We show that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$; by Lemma 27, it suffices to prove that $\mathsf{Failures}(\mathsf{imp}) \subseteq \mathsf{Failures}(\mathsf{spec})$.

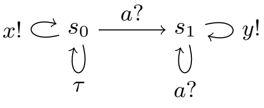
Towards a contradiction, assume that $\mathsf{Failures}(\mathsf{imp}) \not\subseteq \mathsf{Failures}(\mathsf{spec})$. Pick a failure $(w, X) \in \mathsf{Failures}(\mathsf{imp})$ such that $(w, X) \notin \mathsf{Failures}(\mathsf{spec})$. Observe that since $\mathsf{Traces}(\mathsf{imp}) \subseteq \mathsf{Traces}(\mathsf{spec})$, $w \in \mathsf{Traces}(\mathsf{imp}) \cap \mathsf{Traces}(\mathsf{spec})$. Without loss of generality, assume that X is as large as possible: there is no Y such that $(w, Y) \in \mathsf{Failures}(\mathsf{imp}) \setminus \mathsf{Failures}(\mathsf{spec})$ such that $X \subset Y$. Then $\mathsf{imp} \overset{w}{\Longrightarrow} t$ such that $\mathsf{stable}(t)$ holds true and $\mathsf{init}(t) \cap X = \emptyset$.

Note that since imp is an IOTS and t is stable, we have $\mathsf{Act}_I \subseteq \mathsf{init}(t)$ so $X \subseteq \mathsf{Act}_U$. Because imp ioco spec, we have $\mathsf{out}(t) \subseteq \mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w) \subseteq \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$. So there must be a state $s \in \mathsf{spec}\ \mathsf{after}\ w$ such that $\mathsf{out}(t) \cap \mathsf{out}(s) \neq \emptyset$. Let s be such a state, and pick some $x \in \mathsf{out}(t) \cap \mathsf{out}(s)$. Since spec is convergent, all τ-paths are finite and end in a stable state. Because spec is ultimately determined there must be some stable state $s' \in s\ \mathsf{after}\ \epsilon$ such that $\mathsf{out}(s') = \{x\}$. Then $\mathsf{out}(s') \subseteq \mathsf{out}(t)$, and consequently, $\mathsf{init}(s') \cap X = \emptyset$. But then also $(w, X) \in \mathsf{Failures}(\mathsf{spec})$. Contradiction, so $\mathsf{Failures}(\mathsf{imp}) \subseteq \mathsf{Failures}(\mathsf{spec})$ and therefore $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. ◀

Note that there is a rather straightforward reason why we cannot simply drop the assumption on the specification being convergent; see the example below.

▶ **Example 31.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).





Observe that imp ioco spec. Moreover, spec is trivially ultimately determined: s_0 after $\epsilon = \{s_0\}$ and the only output action enabled in s_0 is x. Because s_0 is not stable, we have $(\epsilon, \{y\}) \notin \mathsf{Failures}(s_0)$. On the other hand, $(\epsilon, \{y\}) \in \mathsf{Failures}(i_0)$, so we cannot have $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. ⌟

We finish this section with a similar statement for the internal choice testing theory.

▶ **Proposition 32.** *For any internal choice IOTS* imp *and convergent, ultimately determined internal choice IOTS* spec *satisfying* imp ioco spec *we have* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$.

Proof. Let imp be an internal choice IOTS and spec a convergent, determined internal choice IOTS. Assume that imp ioco spec holds true. We argue that also $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ holds true. Towards a contradiction, suppose that $\mathsf{spec} \not\sqsubseteq_F \mathsf{imp}$. Then, by Lemma 27, $\mathsf{Failures}(\mathsf{imp}) \not\subseteq \mathsf{Failures}(\mathsf{spec})$.

Suppose $\mathsf{Failures}(\mathsf{imp}) \not\subseteq \mathsf{Failures}(\mathsf{spec})$. Pick a failure $(w, X) \in \mathsf{Failures}(\mathsf{imp})$ such that $(w, X) \notin \mathsf{Failures}(\mathsf{spec})$. Then $\mathsf{imp} \overset{w}{\Longrightarrow} t$ such that $\mathsf{stable}(t)$ holds true and $\mathsf{init}(t) \cap X = \emptyset$. Note that since imp is an internal choice IOTS and $\mathsf{stable}(t)$ holds true, we have either $\mathsf{init}(t) = \mathsf{Act}_I$ or $\emptyset \subset \mathsf{init}(t) \subseteq \mathsf{Act}_U$.

- Suppose that $\mathsf{init}(t) = \mathsf{Act}_I$. Because imp is an internal choice IOTS, $\delta \in \mathsf{out}(t)$ and therefore $\delta \in \mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w)$. Since imp ioco spec, also $\delta \in \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$ and hence $w\,\delta \in \mathsf{STraces}(\mathsf{spec})$. This means that there must be some state s such that $\mathsf{spec} \overset{w}{\Longrightarrow} s$, $\mathsf{stable}(s)$ and $\mathsf{init}(s) \cap \mathsf{Act}_U = \emptyset$. Pick such a state s. Since spec is an internal choice IOTS, $\mathsf{init}(s) = \mathsf{Act}_I$. Note that also $\mathsf{init}(t) = \mathsf{Act}_I$ and therefore $\mathsf{init}(s) = \mathsf{init}(t)$. But then also $\mathsf{init}(s) \cap X = \emptyset$. Consequently, $(w, X) \in \mathsf{Failures}(\mathsf{spec})$. Contradiction.
- Suppose that $\emptyset \subset \mathsf{init}(t) \subseteq \mathsf{Act}_U$. Then $\mathsf{Act}_I \subseteq X$. Moreover, because imp ioco spec, we have $\emptyset \subset \mathsf{init}(t) \subseteq \mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w) \subseteq \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$. So, there must be some state s such that $\mathsf{spec} \overset{w}{\Longrightarrow} s$ and $\mathsf{init}(t) \cap \mathsf{out}(s) \neq \emptyset$. Let s be such a state. Since spec is ultimately determined, we find that for all $x \in \mathsf{out}(s)$, there must be some $s' \in s$ after ϵ such that $\mathsf{out}(s') = \{x\}$. Pick some $x \in \mathsf{init}(t) \cap \mathsf{out}(s)$, and let s' be such that $s' \in s$ after ϵ and $\mathsf{out}(s') = \{x\}$. This means that $\mathsf{out}(s') \subseteq \mathsf{init}(t)$. Since spec is convergent and ultimately determined, we may assume that s' is stable. Observe that s' cannot be quiescent since $\mathsf{out}(s') \subseteq \mathsf{init}(t) \subseteq \mathsf{Act}_U$. Since $\mathsf{spec} \in \mathcal{IOTS}^{\sqcap}$, we therefore find that $\mathsf{Act}_I \cap \mathsf{init}(s') = \emptyset$, and hence $\mathsf{init}(s') \subseteq \mathsf{init}(t)$. Consequently, $\mathsf{init}(s') \cap X \subseteq \mathsf{init}(t) \cap X = \emptyset$. From this, we can conclude that $(w, X) \in \mathsf{Failures}(\mathsf{spec})$. Contradiction.

Hence, $\mathsf{Failures}(\mathsf{imp}) \subseteq \mathsf{Failures}(\mathsf{spec})$, and therefore $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. ◀

5 A Small Experiment: Testing Dezyne using mCRL2

As a practical validation of our theory, we apply MBT to a specification and implementation stemming from an industrial model of a multi-component controller at Philips Image Guided Therapy systems. The implementation has been generated from specifications in the Dezyne formal modelling DSL [21, 21]. In the Dezyne development methodology, a system is described as a hierarchical composition of components by specifying:

- a set of behavioural contracts, called *interfaces*. Each interface provides an abstraction of a component, the so-called *provided interface* of the component, and
- a behavioural model (a state machine) that describes how a component realises its behavioural contract, by interacting with subcomponents. The ports via which the component connects to subcomponents are called *required ports*, and by association, the behavioural contracts upon which the component relies are therefore referred to as *required interfaces*.

Table 1 test run results of MBT applied to correct and faulty code-generated implementation.

	correct	Mutation					
average	impl.	1	2	3	4	5	6
detection rate	0%	96%	100%	100%	100%	100%	100%
actions required	200	45	41	9	8	10	17
state coverage	96%	80%	85%	46%	46%	63%	53%

The formal check that takes place in Dezyne, before generating code, is whether a component complies to its provided interface. This check is answered by verifying whether the IOLTS induced by the provided interface is stable-failures refined by the IOLTS obtained by combining the IOLTS underlying the component and the IOLTSs underlying the behavioural contracts of the subcomponents. The actual stable-failures refinement check is conducted using the mCRL2 toolset [2, 8]. In case a component is found to comply to its provided interface (and only then), the behavioural model of the component is fully automatically converted into an equivalent executable C++ program. This way, a correct-by-construction system can be built from the ground-up, or top-down by specifying, in a step-wise manner, desired provided interfaces and introducing (sub)components that "implement" these.

For the system that we study in this section, we do not have access to the implementation of the subcomponents for the required interfaces of our component, but we do have access to their behavioural contracts and the code that was generated from the main component itself. In our experiments, we therefore mimic the behaviour of the subcomponents via a simulator that utilises the IOLTSs of the behavioural contracts of the subcomponents instead. This yields so-called smart stubs. The specification IOLTS of the multi-component controller consists of 25 unique states and 54 unique transitions and is stable-failures testable. As per our theory, the MBT algorithm should not find any non-conformance since the implementation (the component together with the smart stubs) is a stable-failures refinement of the specification (the provided interface). Hence, if a non-conformance is found, the implementation does not reflect the model of the component that was proved to comply to its behavioural contract, and the non-conformance thus signals an actual issue with the executable or the platform.

We are interested in assessing whether we can detect erroneous implementations of the specification using ioco-based MBT techniques. To this end, we test the correct implementation and, in addition, 6 manually created, faulty mutants thereof. The first five faulty mutants are obtained by altering the implementation of the component such that a single randomly chosen input which would normally result in a state change, now performs no actual code execution, and thus results in no state change in the implementation. For the sixth mutant, each provided interface has been given a preset (1/10) chance of remaining idle, instead of providing a response when triggered, which should result in a non-conforming quiescence observation.

Using an on-the-fly MBT algorithm, which implements the original ioco test algorithm [17, 18] in mCRL2, we generated and executed 100 test runs, each consisting of up-to 200 observable actions (including quiescence) for each mutant and for the correct implementation. The results of this experiment are shown in Table 1. For each set of 100 test runs, we measured the percentage of runs that detected a non-conformance, the average number of observable actions (including quiescence) required to observe that non-conformance or terminate (in the case that no non-conformance is detected) and the average specification state coverage, i.e., unique states visited during a test-run. We observe that no non-conformances were detected when testing the correct implementation. In virtually all of the test runs on incorrect implementations a non-conformance was detected when using incorrect implementations, once more confirming the practical relevance of automated testing.

6 Conclusions

We studied the stable-failures refinement relation [15] and its relation to the ioco conformance testing relation by Tretmans [17, 18]. In particular, we identified a set of experiments – called *stable-failures testable traces* – derivable from a specification, for which ioco does not falsely flag implementations as incorrect when these implementations have been shown to refine the specification, thus addressing a major obstacle in applying Model-Based Testing techniques in the Model-Driven Engineering development method. Furthermore, we showed that for internal choice input output transition systems, these experiments coincide with the full set of experiments usually associated with the ioco testing theory. To better understand the limitations of ioco-based testing, we additionally identify conditions under which exhaustive testing can establish that the implementation refines the specification used for testing.

We did not explore how to implement our testing theory efficiently for specifications whose stable-failures testable traces are a proper subset of the suspension traces; this is left for future work. For finite specifications, deriving stable-failures testable traces is easily achieved by means of a determinisation-like algorithm, constructing a *Suspension Automaton* [17, 27] and exploring that structure. For infinite specifications, efficiently deriving and selecting such stable-failures testable traces *on-the-fly* would allow to combine the testing methodology with other *on-the-fly* testing algorithms.

References

1 James Baxter, Ana Cavalcanti, Maciej Gazda, and Robert M. Hierons. Testing using CSP models: Time, inputs, and outputs. *ACM Trans. Comput. Log.*, 24(2):17:1–17:40, 2023. doi:10.1145/3572837.

2 Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *TACAS (2)*, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019. doi:10.1007/978-3-030-17465-1_2.

3 Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in CSP. In *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 2007. doi:10.1007/978-3-540-76650-6_10.

4 Ana Cavalcanti and Robert M. Hierons. Testing with inputs and outputs in CSP. In *FASE*, volume 7793 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2013. doi:10.1007/978-3-642-37057-1_26.

5 Ana Cavalcanti, Robert M. Hierons, and Sidney C. Nogueira. Inputs and outputs in CSP: A model and a testing theory. *ACM Trans. Comput. Log.*, 21(3):24:1–24:53, 2020. doi:10.1145/3379508.

6 Cocotec. Coco platform. https://cocotec.io/, 2023. Accessed: 01 May 2023.

7 Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995. doi:10.1007/3-540-59293-8_188.

8 Jan Friso Groote, Jeroen J. A. Keiren, Bas Luttik, Erik P. de Vink, and Tim A. C. Willemse. Modelling and analysing software in mCRL2. In *FACS*, volume 12018 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2019. doi:10.1007/978-3-030-40914-2_2.

9 Ramon Janssen, Frits W. Vaandrager, and Jan Tretmans. Relating alternating relations for conformance and refinement. In *IFM*, volume 11918 of *Lecture Notes in Computer Science*, pages 246–264. Springer, 2019. doi:10.1007/978-3-030-34968-4_14.

10 Maurice Laveaux, Jan Friso Groote, and Tim A. C. Willemse. Correct and efficient antichain algorithms for refinement checking. *Log. Methods Comput. Sci.*, 17(1), 2021. `doi:10.23638/LMCS-17(1:8)2021`.

11 Sidney C. Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. *Formal Aspects Comput.*, 26(3):441–490, 2014. `doi:10.1007/s00165-012-0258-z`.

12 Neda Noroozi, Ramtin Khosravi, Mohammad Reza Mousavi, and Tim A. C. Willemse. Synchronizing asynchronous conformance testing. In *SEFM*, volume 7041 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2011. `doi:10.1007/978-3-642-24690-6_23`.

13 Neda Noroozi, Ramtin Khosravi, Mohammad Reza Mousavi, and Tim A. C. Willemse. Synchrony and asynchrony in conformance testing. *Softw. Syst. Model.*, 14(1):149–172, 2015. `doi:10.1007/s10270-012-0302-8`.

14 Jan Peleska, Wen-ling Huang, and Ana Cavalcanti. Finite complete suites for CSP refinement testing. *Sci. Comput. Program.*, 179:1–23, 2019. `doi:10.1016/j.scico.2019.04.004`.

15 A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010. `doi:10.1007/978-1-84882-258-0`.

16 Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2008. `doi:10.1007/978-3-540-85778-5_18`.

17 Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 1999. `doi:10.1007/3-540-48320-9_6`.

18 Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. `doi:10.1007/978-3-540-78917-8_1`.

19 Jan Tretmans and Ramon Janssen. Goodbye ioco. In *A Journey from Process Algebra via Timed Automata to Model Learning*, volume 13560 of *Lecture Notes in Computer Science*, pages 491–511. Springer, 2022. `doi:10.1007/978-3-031-15629-8_26`.

20 Rutger van Beusekom, Bert de Jonge, Paul F. Hoogendijk, and Jan Nieuwenhuizen. Dezyne: Paving the way to practical formal software engineering. In *F-IDE@NFM*, volume 338 of *EPTCS*, pages 19–30, 2021. `doi:10.4204/EPTCS.338.4`.

21 Rutger van Beusekom, Jan Friso Groote, Paul F. Hoogendijk, Robert Howe, Wieger Wesselink, Rob Wieringa, and Tim A. C. Willemse. Formalising the Dezyne modelling language in mCRL2. In *FMICS-AVoCS*, volume 10471 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2017. `doi:10.1007/978-3-319-67113-0_14`.

22 Petra van den Bos and Mariëlle Stoelinga. Tester versus bug: A generic framework for model-based testing via games. In *GandALF*, volume 277 of *EPTCS*, pages 118–132, 2018. `doi:10.4204/EPTCS.277.9`.

23 Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *FATES*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2003. `doi:10.1007/978-3-540-24617-6_7`.

24 Martin Weiglhofer. *Automated Software Conformance Testing.* PhD thesis, Graz University of Technology, 2009.

25 Martin Weiglhofer and Bernhard K. Aichernig. Unifying input output conformance. In *UTP*, volume 5713 of *Lecture Notes in Computer Science*, pages 181–201. Springer, 2008. `doi:10.1007/978-3-642-14521-6_11`.

26 Martin Weiglhofer and Franz Wotawa. Asynchronous input-output conformance testing. In *COMPSAC (1)*, pages 154–159. IEEE Computer Society, 2009. `doi:10.1109/COMPSAC.2009.194`.

27 Tim A. C. Willemse. Heuristics for ioco-based test-based modelling. In *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2006. `doi:10.1007/978-3-540-70952-7_9`.

Process-Algebraic Models of Multi-Writer Multi-Reader Non-Atomic Registers

Myrthe S. C. Spronck
Eindhoven University of Technology, The Netherlands

Bas Luttik
Eindhoven University of Technology, The Netherlands

Abstract

We present process-algebraic models of multi-writer multi-reader safe, regular and atomic registers. We establish the relationship between our models and alternative versions presented in the literature. We use our models to formally analyse by model checking to what extent several well-known mutual exclusion algorithms are robust for relaxed atomicity requirements. Our analyses refute correctness claims made about some of these algorithms in the literature.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Theory of computation → Verification by model checking

Keywords and phrases mutual exclusion, model checking, non-atomic reads and writes, regular register

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.5

Related Version *Full Version*: https://arxiv.org/abs/2307.05143

Supplementary Material *Model*: https://github.com/mCRL2org/mCRL2/tree/master/examples/academic/non-atomic_registers
archived at swh:1:dir:f43fbf368f800067a33124501bb01c27b8a9bfa3

Acknowledgements We thank Rob van Glabbeek for insightful discussions on the topic of this paper.

1 Introduction

The mutual exclusion problem was first outlined by Dijkstra [9]. Given n threads executing some code with a special section called the “critical section”, the problem is to ensure that at any one time at most one of the threads is executing its critical section. Dijkstra explicitly states that communication between threads should be done through shared registers, and that reading from and writing to these registers should be considered atomic operations; when two threads simultaneously interact with the register, be it through reading or writing, the register behaves as though these operations took place in some total order.

Lamport argued that solutions to the mutual exclusion problem that assume atomicity of register operations do not fundamentally solve it [19]. After all, implementing atomic operations would require some form of mutual exclusion at a lower level. Many algorithms have been proposed that solve the mutual exclusion problem without requiring atomicity of register operations, most famously Lamport’s own Bakery algorithm [18].

Analysing distributed algorithms using non-atomic registers for communication between threads can be difficult, and correctness proofs are error-prone. Due to the vast number of execution paths of distributed algorithms, especially when overlapping register operations need to be taken into account, manual correctness proofs are likely to miss issues. One better uses computer tools (e.g., model checkers or theorem provers) to support correctness claims with a detailed and preferably exhaustive analysis. This introduces the need for formal models of non-atomic registers.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 5; pp. 5:1–5:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Lamport proposed a general mathematical formalism for reasoning about the behaviour of concurrent systems that do not rely on the atomicity of operations, which he then uses to analyse the correctness of four solutions to the mutual exclusion problem not relying on atomicity [19, 20]. In [21], he studies in more detail the notion of single-writer multi-reader (SWMR) non-atomic register to implement communication between concurrent threads of computation; there, he distinguishes two variants, which he refers to as *safe* and *regular*. When a read operation to a SWMR safe register does not overlap with any write operations, then it will return the value stored in the register, but when it does overlap with a write operation then it may return a completely arbitrary value in the domain of the register. A SWMR regular register is a bit less erratic in the sense that a read operation overlapping with write operations will at least return any of the values actually being written. Raynal presented a straightforward generalisation of the notion of SWMR safe register to the multi-writer case [28]. How the notion of SWMR regular register should be generalised to the multi-writer case, however, is less obvious. Shao et al. discuss four possibilities [29].

The formalisms in [21, 28, 29] for studying the behaviour of non-atomic registers are not directly amenable for analysing the correctness of distributed algorithms by explicit-state model checking, e.g., using the mCRL2 toolset [7]. In fact, it is not clear whether the four variants of MWMR regular registers presented in [29] will lead to a finite-state model even if the number of readers and writers and the set of data values of the register are finite. In [23], Lamport demonstrates a method of modelling SWMR safe registers through repeatedly writing arbitrary values before settling on the desired value, but this approach does not generalise to multi-writer registers. The main contribution of this paper is to present process-algebraic models of multi-writer multi-reader safe, regular and also atomic registers that can be directly used in mCRL2 to analyse the correctness of distributed algorithms.

We have used our process-algebraic models to analyse to what extent various mutual exclusion algorithms are robust for relaxed non-atomicity requirements. We find that Peterson's algorithm [27] no longer guarantees mutual exclusion if the atomicity requirement is relaxed for the turn register. A variant of Peterson's algorithm presented in [4] does guarantee mutual exclusion even if registers are only safe. The variant presented in [29], however, does not guarantee mutual exclusion with regular registers, despite a claim that it does. We also find that some of the algorithms proposed in [31, 32] do not guarantee mutual exclusion for regular registers, which seems to contradict claims that they are immune to the problem of flickering bits during writes. When analysing Lamport's 3-bit algorithm [20] we discovered that its mutual exclusion guarantee crucially depends on how one of the more complex statements of the algorithm is implemented. Finally, we confirm that Aravind's BLRU algorithm [3], Dekker's algorithm [1], Dijkstra's algorithm [9] and Knuth's algorithm [17] guarantee mutual exclusion even with safe registers.

This paper is organised as follows. In Section 2 we present some basic definitions pertaining to SWMR registers, including formalisations of Lamport's notions of SWMR safe, regular and atomic registers. In Section 3 we present and discuss our process-algebraic definitions of MWMR safe, regular and atomic registers, and establish formal relationships with their SWMR counterparts. In Section 4 we compare our notion of MWMR regular register with the variants of MWMR regular registers proposed by [29]. In Section 5 we report on our analyses of the various mutual exclusion algorithms. Finally, we present conclusions and some ideas for future work in Section 6.

2 Single-writer multi-reader registers

The definitions presented in this section are adapted from [29] and [22].

We consider n threads operating on a register with values in a finite set $\mathbb{D}$ of register values; the initial value of the register will be denoted by d_{init}. Threads are identified by a natural number in the set $\mathbb{T} = \{0, \ldots, n-1\}$. A *read operation* by thread $i \in \mathbb{T}$ on the register, with *return value* $d \in \mathbb{D}$, is a sequence $r_i(d) = sr_i fr_i(d)$ consisting of an *invocation* sr_i (for "thread i starts to read"), and a matching *response* $fr_i(d)$ (for: "the read by thread i finishes with return value d"). A *write* operation of thread i on the register, with *write value* d, is a sequence $w_i(d) = sw_i(d)fw_i$ consisting of an *invocation* $sw_i(d)$ (for: "thread i starts to write value d") and a matching *response* fw_i (for: "the write by thread i finishes"). An *operation* of thread i is either a read operation or a write operation of that thread.

For every $i \in \mathbb{T}$, let $A_i = \{sr_i, fr_i(d), sw_i(v), fw_i \mid d \in \mathbb{D}\}$, and let $A = \bigcup_{i\in\mathbb{T}} A_i$. If σ is a sequence of elements of A, then we denote by $\sigma|i$ the subsequence of σ consisting of the elements in A_i. A *schedule* on a register is a finite or infinite sequence σ of elements of A such that $\sigma|i$ consists of alternating invocations and matching responses, beginning with an invocation, and if $\sigma|i$ is finite, ending with a response. Note that, by these requirements and our definition of the notion of operation, $\sigma|i$ can then be obtained as the concatenation of read and write operations $o_0o_1o_2\ldots$ executed by thread i.[1] We shall denote by $ops(\sigma, i)$ the set of all operations executed by thread i (i.e., $ops(\sigma, i) = \{o_0, o_1, o_2, \ldots\}$) and by $ops(\sigma)$ the set of all operations executed by any of the threads. It is technically convenient to include in $ops(\sigma)$ a special write operation w_{init}that writes the initial value of the register. Then $ops(\sigma) = \{w_{init}\} \cup \bigcup_{i\in\mathbb{T}} ops(\sigma, i)$. We also use $reads(\sigma)$ and $writes(\sigma)$ for the subsets of $ops(\sigma)$ respectively consisting of the read operations and the write operations only.

A schedule σ induces a partial order on $ops(\sigma)$: if $o, o' \in ops(\sigma)$, then we write $o <_\sigma o'$ if, and only if, the response of o precedes the invocation of o' in σ. We stipulate that $w_{init} < o$ for all $o \in ops(\sigma)\backslash\{w_{init}\}$. Let $r \in ops(\sigma)$ be a read operation and let $w \in ops(\sigma)$ be a write operation. We say that w is *fixed* for r if $w <_\sigma r$; *fix-writes*(σ, r) denotes the set of all writes that are fixed for r. We say that w is *relevant* for r if $r \not<_\sigma w$; *rel-writes*(σ, r) denotes the set of all writes in $ops(\sigma)$ that are relevant for r. Note that, by the inclusion of w_{init}, the sets *rel-writes*(σ, r) and *fix-writes*(σ, r) are non-empty for all $r \in reads(\sigma)$. We say that $r \in reads(\sigma)$ *can read from* $w \in writes(\sigma)$ if w is relevant for r and there does not exist $w' \in writes(\sigma)$ such that $w <_\sigma w' <_\sigma r$. An operation o has *overlapping writes* if there exists $w \in writes(\sigma)$ such that $o \not<_\sigma w$ and $w \not<_\sigma o$.

In [29], a register model is defined as a set of schedules satisfying certain conditions. Restricting attention to single-writer multi-reader (SWMR) registers only, Lamport considers three register models: safe, regular and atomic [22]. We proceed to define Lamport's models by formulating conditions on *single-writer* schedules, i.e., schedules in which all write operations are by one particular thread. If σ is a single-writer schedule, then, since a write cannot have overlapping writes, every non-empty finite set W of writes has a $<_\sigma$-maximum, i.e., an element $w \in W$ such that $w' <_\sigma w$ for all $w' \in W \setminus \{w'\}$. Since writes that are fixed for r have their responses in the finite prefix of σ preceding the invocation of r, we have that *fix-writes*(σ, r) is finite for every r. Since *fix-writes*(σ, r) is non-empty, it always has a $<_\sigma$-maximum.

[1] The same operation may occur multiple times in $\sigma|i$. Henceforth, when we consider an operation in $\sigma|i$ we actually mean to refer to a specific occurrence in $\sigma|i$ of the operation. To disambiguate between two different occurrences of the same operation o we could, e.g., annotate each occurrence of o with its position in $\sigma|i$. We will not do so explicitly, because it will unnecessarily clutter the presentation. But the reader should keep in mind that, whenever we refer to an operation in a schedule σ we actually mean to refer to a particular occurrence of that operation in $\sigma|i$.

A SWMR register is *safe* if a read that does not have overlapping writes returns the most recently written value. A read that does have overlapping writes may return any arbitrary value in the domain $\mathbb{D}$ of the register.

▶ **Definition 1.** *A single-writer schedule σ is* safe *if every read r without overlapping writes returns the value written by the $<_\sigma$-maximum of the set of fix-writes(σ, r).*

A SWMR register is *regular* if it is safe, and a read that has overlapping writes returns the value of one of the overlapping writes or the most recently written value.

▶ **Definition 2.** *A single-writer schedule σ is* regular *if every read r returns either the value written by the $<_\sigma$-maximum of the set fix-writes(σ, r) or the value of an overlapping write.*

A SWMR register is *atomic* if all reads and writes behave as though they occur in some definite order. A *serialisation* is a total order $\mathcal{S}$ on a subset O of $ops(\sigma)$ that is *consistent* with $<_\sigma$ in the sense that for all $o, o' \in O$ we have that $o <_\sigma o'$ implies $o \mathrel{\mathcal{S}} o'$. A serialisation $(O, \mathcal{S})$ is *legal* if every read operation returns the value of the most recent write operation according to $\mathcal{S}$, that is, whenever $r \in O$ is a read operation with return value v, then v is the write value of $\mathcal{S}$-maximum of *rel-writes*(σ, r).

▶ **Definition 3.** *A single-writer schedule σ is* atomic *if $ops(\sigma)$ has a legal serialisation.*

3 Multi-writer multi-reader registers

We now want to define multi-write multi-reader (MWMR) safe, regular and atomic registers. Since our goal is to verify the correctness of mutual exclusion algorithms by model checking, we prefer operational, process-algebraic definitions of register models over definitions in terms of schedules. We are going to define register models by giving recursive process definitions that, given the state of the register, admit certain interactions with the register, resulting in an update of the state of the register. Which information needs to be maintained in the state of the register depends on the register model, but the state of register should at least reflect which operations are currently active. So, with each register model $m \in \{s, r, a\}$ we associate a set of states $\mathbb{S}_m$, and we assume that the following functions are defined on $\mathbb{S}_m$:

$$\begin{aligned}
&\mathit{rdrs}, \mathit{wrtrs}, \mathit{idle} : \mathbb{S}_m \to \mathcal{P}(\mathbb{T}) \\
&\mathit{usr}_i, \mathit{ufr}_i, \mathit{ufw}_i : \mathbb{S}_m \to \mathbb{S}_m \\
&\mathit{usw}_i : \mathbb{D} \times \mathbb{S}_m \to \mathbb{S}_m \quad .
\end{aligned} \tag{1}$$

The mappings *rdrs* returns the set of all threads that are currently reading, i.e., $i \in \mathit{rdrs}(s)$ if, and only if, thread i has invoked a read operation but the matching response has not yet occurred. Similarly, *wrtrs* returns the set of all threads that are currently writing, and *idle* returns the set of all threads that are currently not reading and not writing. The mappings usr_i, ufr_i, usw_i and ufw_i perform update operations on the state of the register, corresponding to whether the most recent interaction of the register was an invocation (usr_i) or response (ufr_i) of a read, or an invocation (usw_i) or a response (ufw_i) of a write. The update operation usw_i also takes the write value into account.

In the remainder of this section we shall first present our models of MWMR safe, regular and atomic registers, and then comment on the representation of these models in mCRL2.

3.1 MWMR Safe Registers

Lamport's SWMR safe register model (see Definition 1) accounts for how reads and writes behave when they do not have overlapping writes, and how reads behave when they do have overlapping writes. To generalise Lamport's notion to MWMR registers, we need to define

$$
\begin{array}{l}
R_s(d : \mathbb{D}, s : \mathbb{S}_s) = \\
\qquad \displaystyle\sum_{i \in \mathbb{T}} \left(\begin{array}{ll}
 & (i \in \mathit{idle}(s)) \to \mathit{sr}_i \cdot R_s(d, \mathit{usr}_i(s)) \\
+ & (i \in \mathit{idle}(s)) \to \sum_{d' \in \mathbb{D}} \mathit{sw}_i(d') \cdot R_s(d, \mathit{usw}_i(d', s)) \\
+ & (i \in \mathit{rdrs}(s) \wedge \neg \mathit{overlap}_i(s)) \to \mathit{fr}_i(d) \cdot R_s(d, \mathit{ufr}_i(s)) \\
+ & (i \in \mathit{rdrs}(s) \wedge \mathit{overlap}_i(s)) \to \sum_{d' \in \mathbb{D}} \mathit{fr}_i(d') \cdot R_s(d, \mathit{ufr}_i(s)) \\
+ & (i \in \mathit{wrtrs}(s) \wedge \neg \mathit{overlap}_i(s)) \to \mathit{fw}_i \cdot R_s(\mathit{next}(s), \mathit{ufw}_i(s)) \\
+ & (i \in \mathit{wrtrs}(s) \wedge \mathit{overlap}_i(s)) \to \sum_{d' \in \mathbb{D}} \mathit{fw}_i \cdot R_s(d', \mathit{ufw}_i(s))
\end{array} \right)
\end{array}
$$

Figure 1 Safe register model.

how writes behave when they have overlapping writes. We follow Raynal's approach and define that when a write has overlapping writes, then its effect is that some arbitrary value in $\mathbb{D}$ is written to the register [28].

Our process-algebraic definition of a MWMR safe register is shown in Figure 1. The equation defines the behaviour of processes $R_s(d, s)$; the parameter $d \in \mathbb{D}$ reflects the current value of the register, and the parameter $s \in \mathbb{S}_s$ reflects its current state. For the behaviour of the safe register it must be determined for every read or write operation of a thread whether, during its interaction with the register, there was an overlapping write operation by some other thread. Therefore, in addition to the functions specified in Equation 1, we presuppose on $\mathbb{S}_s$ a predicate $\mathit{overlap}_i$ such that $\mathit{overlap}_i(s)$ holds if during the interaction of thread i with the register there was an overlapping write by another thread. At the response of a write that is not overlapping with other writes, the current value d of the register needs to be replaced by the write value. Hence, whenever a write is invoked, the write value is stored in s through $\mathit{usw}_i(s)$; this value can be retrieved with the mapping $\mathit{next} : \mathbb{S}_s \to \mathbb{D}$ if the write had no overlapping writes. If there were overlapping writes, next is undefined. The right-hand side of the equation in Figure 1 specifies the behaviour of the register using standard process-algebraic operations: $\cdot$ denotes sequential composition, $+$ denotes non-deterministic choice, $\to$ denotes a conditional, and $\sum$ denotes choice quantification [14].

The definition in Figure 1 induces a transition relations $\xrightarrow{a}$ ($a \in A$) on the set of tuples $\langle d, s \rangle$ ($d \in \mathbb{D}$, $s \in \mathbb{S}_s$). For instance, if $i \in \mathit{rdrs}(s)$ and $\neg\mathit{overlap}_i(s)$, then there is a transition

$$\langle d, s \rangle \xrightarrow{\mathit{fr}_i(d)} \langle d, \mathit{ufr}_i(s) \rangle \ ,$$

according to the third summand of the definition in Figure 1; and if $i \in \mathit{wrtrs}(s)$ and $\mathit{overlap}_i(s)$, then, for every $d' \in \mathbb{D}$, there is a transition

$$\langle d, s \rangle \xrightarrow{\mathit{fw}_i} \langle d', \mathit{usw}_i(s) \rangle \ ,$$

according to the last summand of the definition in Figure 1.

We let s_{init} denote the *initial state* of the safe register, and we define $\mathit{idle}(s_{\mathit{init}}) = \mathbb{T}$, $\mathit{wrtrs}(s_{\mathit{init}}) = \mathit{rdrs}(s_{\mathit{init}}) = \emptyset$, $\mathit{overlap}_i(s)$ is false, and $\mathit{next}(s) = d_{\mathit{init}}$. Henceforth, we shall abbreviate $R_s(d_{\mathit{init}}, s_{\mathit{init}})$ by R_s. A *trace* of R_s is a finite or infinite sequence $a_0 a_1 \cdots a_{n-1} a_n \cdots$ of elements of A such that there exist $d_0, d_1, d_2, \ldots, d_n, \ldots \in \mathbb{D}$ and $s_0, s_1, s_2, \ldots, s_n, \ldots \in \mathbb{S}_s$ with $d_0 = d_{\mathit{init}}$ and $s_0 = s_{\mathit{init}}$ and $\langle d_0, s_0 \rangle \xrightarrow{a_0} \langle d_1, s_1 \rangle \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} \langle d_n, s_n \rangle \xrightarrow{a_n} \cdots$. We denote by $\mathcal{T}_s$ the set of all traces of R_s. A trace $\alpha \in \mathcal{T}_s$ is *complete* if, for all $i \in \mathbb{T}$, either $\alpha | i$ is infinite or $\alpha | i$ ends with a response. A *single-writer* trace is a trace in which all invocations and responses of write operations are by the same thread.

We argue that there is a one-to-one correspondence between the single-writer safe schedules and the single-writer complete traces of R_s. First, note that schedules and complete traces adhere to exactly the same restrictions regarding the order in which invocations and responses of read and write operations can occur: the invocation of an operation by some thread can only occur when that same thread is not currently executing another operation, and a response to some thread for an operation can only occur if the last interaction of that thread was, indeed, an invocation of that same operation. Write values are not restricted in schedules, nor in complete traces. Moreover, in the single-writer case the value of the parameter d of the process R_s will always be the write value of write operation of which the execution finished last. Finally, note that both in schedules and in complete traces of R_s, if a read operation overlaps with a write operation, then it may return any value, and if it does not, then it will, indeed, return the value of the most recent write operation.

▶ **Proposition 4.** *Every single-writer safe schedule is a trace of R_s, and every complete single-writer trace of R_s is a safe schedule.*

3.2 MWMR regular registers

According to Lamport's definition of SWMR regular registers (see Definition 2), a read either returns the write value of the $<_\sigma$-maximum of $\mathit{fix\text{-}writes}(\sigma, r)$ or the value written by one of its overlapping writes. When writes may have overlapping writes, then $\mathit{fix\text{-}writes}(\sigma, r)$ may not have a $<_\sigma$-maximum. It is then necessary to determine, for every read r, which of the $<_\sigma$-maximal elements of $\mathit{fix\text{-}writes}(\sigma, r)$ should be taken into account when determining the return value of r, and to what extent different reads should agree on this choice.

Our considerations are as follows. First, we want our MWMR regular register model to coincide with Lamport's SWMR regular register model when there are no writes overlapping other writes, so that our analyses of algorithms that rely on SWMR regular registers are valid with respect to Lamport's model. Second, our model should be suitable for explicit-state model checking. This precludes any definition that requires keeping track of unbounded information pertaining to the history of the computation. To limit the amount of information that the model is required to remember, we let the register commit to a unique value when there are no active writes. In this respect, our model deviates from three of the four models considered in [29]; in Section 4 we provide a more detailed comparison.

To be consistent with Lamport's SWMR regular registers, a read r should be able return the value of any *overlapping* write. To determine which of the elements of the *fixed* writes is taken into account when determining the return value of r, our model non-deterministically inserts a special *order* action ow_i somewhere between the invocation and the response of every write of every thread $i \in \mathbb{T}$. One may think of the order action as marking the moment at which the write truly takes place. Note that this order action is purely for modelling purposes, we make no claims on the implementation of a regular register. The write value associated with the most recent order action preceding the invocation of a read (or the initial value if no order actions have occurred yet) is taken into account as possible return value for that read. Thus, a serialisation of all writes is generated on-the-fly through the order actions: all read operations agree on the order of the writes.

Our process-algebraic definition of a MWMR regular register is given in Figure 2. Here, $\mathbb{S}_r$ denotes the set of possible states of the MWMR regular register. The register keeps track of the readers, writers and idle threads, similar to the safe register. It additionally keeps track of the set $\mathit{pndng}(s)$ of threads that have invoked a write but for which the order action has not yet occurred. The update function $\mathit{uow}_i : \mathbb{S}_r \to \mathbb{S}_r$ associated with the order action

$$R_r(d:\mathbb{D}, s:\mathbb{S}_r) = \sum_{i\in\mathbb{T}} \left(\begin{array}{ll} & (i \in \mathit{idle}(s)) \to \mathit{sr}_i \cdot R_r(d, \mathit{usr}_i(s)) \\ + & (i\in \mathit{idle}(s)) \to \sum_{d'\in\mathbb{D}} \mathit{sw}_i(d')\cdot R_r(d, \mathit{usw}_i(d',s)) \\ + & (i \in \mathit{rdrs}(s)) \to \sum_{d'\in \mathit{pval}_i(s)} \mathit{fr}_i(d') \cdot R_r(d, \mathit{ufr}_i(s)) \\ + & (i\in \mathit{pndng}(s)) \to \mathit{ow}_i \cdot R_r(\mathit{wval}_i(s), \mathit{uow}_i(s)) \\ + & (i\in \mathit{wrtrs}(s) \land i \notin \mathit{pndng}(s)) \to \mathit{fw}_i \cdot R_r(d, \mathit{ufw}_i(s))\end{array}\right)$$

Figure 2 Regular register model.

ow_i removes thread i from $\mathit{pndng}(s)$. For every thread $i \in \mathit{pndng}(s)$, $\mathit{wval}_i(s)$ is the write value of that write; it is used to correctly update the current value d of the register when ow_i occurs. For every thread $i \in \mathit{rdrs}(s)$, $\mathit{pval}_i(s)$ is the set of values that a read r invoked by thread i may return. That is, it consist of the values of all writes overlapping with r (thus far) and the value of the write with the most recent ow_j before the invocation of r.

For $i \in \mathbb{T}$, let $A_i^r = A_i \cup \{\mathit{ow}_i\}$, and let $A^r = \bigcup_{i\in\mathbb{T}} A_i^r$. The process definition in Figure 2 induces transition relations $\xrightarrow{a}$ $(a \in A^r)$ on the set of tuples $\langle d, s\rangle$ $(d \in \mathbb{D},\ s \in \mathbb{S}_r)$. As before $\mathit{idle}(s_{\mathit{init}}) = \mathbb{T}$, $\mathit{rdrs}(s_{\mathit{init}}) = \mathit{wrtrs}(s_{\mathit{init}}) = \emptyset$. We also have $\mathit{pndng}(s_{\mathit{init}}) = \emptyset$, and $\mathit{pval}_i(s_{\mathit{init}}) = \emptyset$ for all $i \in \mathbb{T}$. The initial values for $\mathit{wval}_i(s_{\mathit{init}})$ do not matter, since $\mathit{wval}_i(s)$ only matters when $i \in \mathit{pndng}(s)$. We use R_r to abbreviate $R_r(d_{\mathit{init}}, s_{\mathit{init}})$, and define a trace of R_r, also as before, as a finite or infinite sequence of elements of A^r appearing as labels in a transition sequence starting at $\langle d_{\mathit{init}}, s_{\mathit{init}}\rangle$. We denote by $\mathcal{T}_r$ the set of all traces of R_r.

Compared to schedules, the traces of R_r have extra ow_i actions. If α is a finite or infinite sequence of elements of A^r, then we denote by $\bar{\alpha}$ the sequence of elements of A obtained from α by deleting all occurrences of ow_i $(i \in \mathbb{T})$. We can then formulate a correspondence between the single-writer traces of R_r (i.e., the traces in which all invocations and responses of write operations are by the same thread) and single-writer regular schedules.

If writes have no overlapping writes, then the most recent order action when a read r is invoked either corresponds to the $<_\sigma$-maximum of $\mathit{fix\text{-}writes}(\sigma, r)$, or to a write that overlaps with r. In the first case, the set of possible values that can be returned by the read according to our model will coincide with the set of possible values that it can return according to Definition 2. In the latter case, our model allows a subset of the values possible according to Definition 2 to be returned. Hence, a read in our model never returns a value that could not be returned according to Lamport's SWMR definition of regular registers. Moreover, if there is a trace of R_r in which the order action ow_i of a write that overlaps with r occurs before the invocation of r, then there also exist a trace in which it occurs after the invocation of r. Thus, the set of traces described by our model includes all regular schedules according to Definition 2 whenever there are no writes overlapping other writes.

▶ **Proposition 5.** *For every single-writer regular schedule σ there is a trace α of R_r such that $\bar{\alpha} = \sigma$, and if α is a complete single-writer trace of R_r, then $\bar{\alpha}$ is a regular schedule.*

3.3 MWMR atomic registers

Definition 3, formalising Lamport's notion of SWMR atomic register, straightforwardly generalises to MWMR registers by omitting the single-writer restriction on schedules. Our process-algebraic model should generate the legal serialisation of all operations on-the-fly. To this end, we introduce, for every thread i, *execution* actions er_i and ew_i to mark the exact moment at which an operation is treated as occurring. An operation's execution action must, of course, occur between its invocation and response. The value that is returned at the response of a read is the value that the register stored at the moment of that read's execution; the register's stored value is updated to a write's value at that write's execution.

The process-algebraic model of our MWMR atomic register is shown in Figure 3. The set of states of R_a is denoted by $\mathbb{S}_a$. In addition to the standard update functions, there are extra update functions $uer_i, uew_i : \mathbb{S}_a \to \mathbb{S}_a$ for the execution actions. The effect of applying uer_i on s is to store the current value d of the register as the value that should be returned at the response of the active read by thread i; this value can then be retrieved with $vals_i(s)$, and $vals_i(s) = \bot$ until then. The effect of applying uew_i is to update the current value d of the register to the write value of the active write by thread i; this value can also be retrieved with $vals_i(s)$, and $vals_i(s) = \bot$ thereafter. Note that, by setting $vals_i(s)$ to $\bot$ before a read has been executed and after a write has been executed, we can use $vals_i(s)$ in combination with $rdrs(s)$ and $wrtrs(s)$ to determine whether the execution of an operation has taken place.

$$R_a(d : \mathbb{D}, s : \mathbb{S}_a) =$$
$$\sum_{i \in \mathbb{T}} \left(\begin{array}{ll} & (i \in idle(s)) \to sr_i \cdot R_a(d, usr_i(s)) \\ + & (i \in idle(s)) \to \sum_{d' \in \mathbb{D}} sw_i(d') \cdot R_a(d, usw_i(d', s)) \\ + & (i \in rdrs(s) \wedge vals_i(s) = \bot) \to er_i \cdot R_a(d, uer_i(s)) \\ + & (i \in wrtrs(s) \wedge vals_i(s) \neq \bot) \to ew_i \cdot R_a(vals_i(s), uew_i(s)) \\ + & (i \in rdrs(s) \wedge vals_i(s) \neq \bot) \to fr_i(vals_i(s)) \cdot R_a(d, ufr_i(s)) \\ + & (i \in wrtrs(s) \wedge vals_i(s) = \bot) \to fw_i \cdot R_a(d, ufw_i(s)) \end{array} \right)$$

Figure 3 Atomic register model.

For $i \in \mathbb{T}$, let $A_i^a = A \cup \{er_i, ew_i\}$, and let $A^a = \bigcup_{i \in \mathbb{T}} A_i^a$. The process definition in Figure 3 induces transition relations $\xrightarrow{a}$ ($a \in A^a$) on the set of tuples $\langle d, s \rangle$ ($d \in \mathbb{D}$, $s \in \mathbb{S}_a$). As before $idle(s_{init}) = \mathbb{T}$ and $rdrs(s_{init}) = wrtrs(s_{init}) = \emptyset$; the initial values for $vals_i(s_{init})$ do not matter. We use R_a to abbreviate $R_a(d_{init}, s_{init})$, and define a trace of R_A, also as before, as a finite or infinite sequence of elements of A^A appearing as labels in a transition sequence starting at $\langle d_{init}, s_{init} \rangle$. We denote by $\mathcal{T}_a$ the set of all traces of R_a.

Compared to schedules, the traces of R_a have extra er_i and ew_i actions. If α is a finite or infinite sequence of elements of A^a, then we denote by $\bar{\alpha}$ the sequence obtained from α by deleting all occurrences of er_i and ew_i for $i \in \mathbb{T}$. The correspondence between atomic schedules and complete traces of R_a follows straightforwardly. It suffices to prove that R_a admits exactly those traces α such that there exists a legal serialisation of $\bar{\alpha}$. To this end, note that the execute actions provide such a serialisation, and the definition of R_a has the responses of operations behave in accordance with this serialisation.

▶ **Proposition 6.** *For every atomic schedule σ there is a trace α of R_a such that $\bar{\alpha} = \sigma$, and if α is a complete trace of R_a, then $\bar{\alpha}$ is an atomic schedule.*

3.4 mCRL2 implementation

The mCRL2 toolset [7] provides tools for model checking and equivalence checking. Models are defined in the mCRL2 language [14], which comprises a process-algebraic specification language and facilitates the algebraic specification of data types. Properties defined in the modal μ-calculus can be checked on those models. One nice feature of mCRL2 is that when a property does not hold a counterexample can be generated. For more information we refer to [14] as well as the toolset's website[2].

We have implemented the models presented in Figures 1, 2 and 3 in the mCRL2 language. By adding processes that model the threads executing the desired algorithm in a manner compatible with the interface of the register models, we can verify the same algorithm easily under different atomicity assumptions. An added benefit is that we can assume different levels of atomicity for different registers simultaneously, so that we pinpoint exactly to what extent the algorithm is robust for non-atomicity. The model can be found as part of the examples delivered with the mCRL2 distribution[3].

The mCRL2 language has support for standard data types such as sets, bags and arrays (implemented as mappings) as well an algebraic specification facility to define new datatypes. This allows us to model the register models staying close to the process-algebraic models presented in this paper.

4 Alternative definitions of MWMR regular registers

In [29] four definitions for MWMR regular registers are proposed. These are formulated as conditions on schedules. We discuss how our definition of MWMR regular registers relates to these definitions.

The following definition captures the weakest condition on schedules presented in [29].

▶ **Definition 7.** *A schedule σ satisfies the* weak *condition if, for every read operation r in $ops(\sigma)$, there exists a legal serialisation of $writes(\sigma) \cup \{r\}$.*

It follows straightforwardly from our MWMR regular register definition that any complete trace $\alpha \in \mathcal{T}_r$, when transformed into a schedule $\bar{\alpha}$ by deleting the order actions, satisfies Definition 7. As explained in Section 3.2, our model generates a serialisation of all writes. For every read r by thread i, it returns either the value of the last write in this serialisation before sr_i, or the value of one of the writes overlapping this read. In both cases, we may obtain a legal serialisation of $writes(\bar{\alpha}) \cup \{r\}$ by taking the serialisation of writes associated with $\bar{\alpha}$ and inserting r right after the write that it reads from. This is consistent with $<_\sigma$ because the serialisation of the writes is, and r will only be placed after a write that either has its response before the invocation of r, or that r overlaps with.

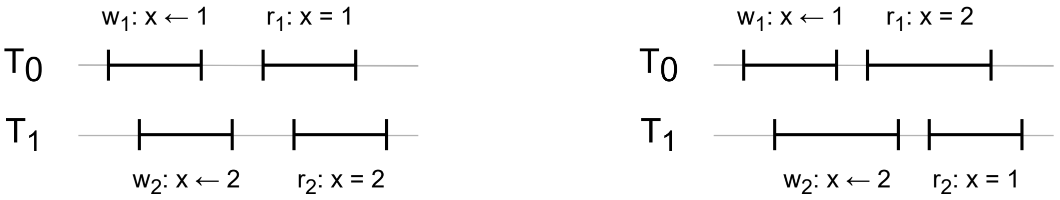
▶ **Proposition 8.** *If $\alpha \in \mathcal{T}_r$ is complete, then the schedule $\bar{\alpha}$ satisfies the weak condition.*

In all our MWMR register definitions it is the case that when no writes are active on a register, it stores a unique value. It reduces the burden of storing elaborate information on the execution history of the register, as would be necessary with the definitions of [29], and thus leads to a smaller statespace. A consequence of our choice is that not all schedules satisfying the weak condition can be generated by our model.

▶ **Example 9.** Consider the schedule depicted in Figure 4a. It is argued in [29, Figure 6] that it satisfies the weak condition, but it cannot be generated by our regular register model R_r because once w_1 and w_2 have ended, the register will have stored a unique value (either 1 or 2). Hence, the return values of r_1 and r_2 cannot be different. Note that, for the same reason, the schedule cannot be generated by our safe register model R_s.

[2] `https://www.mcrl2.org`

[3] `https://github.com/mCRL2org/mCRL2/tree/master/examples/academic/non-atomic_registers` (972629b)



(a) A schedule allowed by the weak, reads-from and no-inversion definitions but not by our regular register model.

(b) A schedule allowed by our regular register model but not by the write-order definition.

Figure 4 Schedules demonstrating the differences between our regular register model and the definitions in [29]. We illustrate these schedules on a timeline, where an operation is drawn from its invocation to its response.

As illustrated in the preceding example, there exist schedules satisfying the weak condition that cannot be generated by our safe register model R_s. Conversely, it is easy to see that there exist complete traces generated by our safe register model R_s (e.g., with overlapping writes resulting in a value that is not written by any of the writes) that do not satisfy the weak condition.

The second condition in [29] associates with every read operation a serialisation and formulates a consistency requirement on these serialisations. If $r \in \mathit{reads}(\sigma)$, then an *$r$-serialisation* is a serialisation $\mathcal{S}_r$ on $\mathit{rel\text{-}writes}(\sigma) \cup \{r\}$.[4]

▶ **Definition 10.** *A schedule σ satisfies* write-order *if for each read r in $\mathit{ops}(\sigma)$ there exists a legal serialisation $\mathcal{S}_r$ of $\mathit{rel\text{-}writes}(\sigma) \cup \{r\}$ satisfying the following condition: for all reads r_1, r_2 in $\mathit{ops}(\sigma)$, and for all writes $w_1, w_2 \in \mathit{rel\text{-}writes}(\sigma, r_1) \cap \mathit{rel\text{-}writes}(\sigma, r_2)$ it holds that $w_1 \; \mathcal{S}_{r_1} \; w_2$ if and only if $w_1 \; \mathcal{S}_{r_2} \; w_2$.*

▶ **Proposition 11.** *For every schedule σ satisfying the write-order condition, there exists a trace α in $\mathcal{T}_r$ such that $\bar{\alpha} = \sigma$.*

We give a brief, informal description of how such a trace α can be constructed here; a more formal argument is presented in [30, Appendix A]. The idea is that order actions can be inserted between the invocation and response of every write in σ, such that the return values of the reads match this placement of order actions. Note that for reads that return the value of an overlapping write, this return value is possible according to Figure 2 regardless of how the order actions are placed. In our placement of order actions, we therefore only need to carefully consider reads that return the value of a write that is fixed for them. According to Definition 10, reads in σ agree on the relative ordering of all writes that are relevant to them. Since $\mathit{fix\text{-}writes}(\sigma, r) \subseteq \mathit{rel\text{-}writes}(\sigma, r)$ for every read r, the reads also agree on the relative ordering of the fixed writes. We use this information to construct an ordering on all writes that is consistent both with $<_\sigma$ and with the return values of reads that read from writes that are fixed for them. Effectively, we find a single view on the relative order of all the write operations that is possible for every read in the schedule that returns the value of a fixed write. Using this ordering, we can then place the order actions in the schedule σ to create the trace $\alpha \in \mathcal{T}_r$ such that $\bar{\alpha} = \sigma$.

[4] By considering serialisations of the relevant writes for r, instead of all writes, we deviate from [29]. Since a serialisation $\mathcal{S}$ on $\mathit{writes}(\sigma) \cup \{r\}$ must be consistent with $<_\sigma$, we will have that $r \; \mathcal{S} \; w$ for all $w \in \mathit{writes}(\sigma) \setminus \mathit{rel\text{-}writes}(\sigma)$. It follows that the restriction of a serialisation $\mathcal{S}$ on $\mathit{writes}(\sigma) \cup \{r\}$ to $\mathit{rel\text{-}writes}(\sigma) \cup \{r\}$ is an r-serialisation, and $\mathcal{S}$ is legal if, and only if, its restriction is.

Whilst every schedule satisfying Definition 10 corresponds to a trace of our model, not every schedule with a corresponding trace in our model is allowed by the write-order condition.

▶ **Example 12.** Consider Figure 4b. This schedule is allowed by our model; r_1 can read 2 in x because it overlaps with w_2 and it is possible for r_2 to read 1 if the order action of w_2 is done before the order action of w_1. This schedule does not meet Definition 10 however; since both writes to x are relevant for both reads, the two reads must agree on the respective order of the writes. For r_2 to read 1, it must be the case that $w_2 \mathcal{S}_{r_2} w_1$. But since $w_1 < r_1$ according to the schedule, this means that $w_2 \mathcal{S}_{r_1} w_1 \mathcal{S}_{r_1} r_1$, so r_1 cannot read 2.

The third and fourth conditions on schedules proposed in [29] we refer to as *reads-from* [29, Definition 9] and *no-inversion* [29, Definition 10], respectively. We do not recall these conditions here, and instead refer to [29] for more details.

Our notion of MWMR regular register is incomparable with the notions induced by the reads-from and no-inversion conditions on schedules. First, as already indicated, every schedule that satisfies the write-order condition is also allowed by our model. As it is proven in [29] that the write-order condition is incomparable with the reads-from and no-inversion conditions, this means our model admits schedules not admitted by these definitions. To see that that not all schedules satisfying reads-from and no-inversion are admitted by our model, it suffices to observe that the schedule presented in Figure 4a, which is not admitted by our MWMR regular register model, satisfies the reads-from and the no-inversion conditions. (See, e.g., [29, Figure 8] and [29, Figure 9], which satisfy the reads-from and no-inversion conditions, respectively, and have the schedule in Figure 4a as prefix.)

5 Verifying Mutual Exclusion Protocols

We have used the register processes described in Section 3 to analyse several well-known mutual exclusion algorithms. To this end, we have modelled the behaviour of the threads as prescribed by the algorithm also as processes, which interact with the register processes. That a thread is executing its non-critical section is represented in our model by the action *noncrit*, and that is executing its critical section is represented by the action *crit*; both actions are parameterised with the thread id. We have checked the following two properties.

▶ Property 1 (Mutex). There is no state reachable from the initial state of the model in which there are two distinct threads i and j such that $crit(i)$ and $crit(j)$ are both enabled in this state.

▶ Property 2 (Reach). For all threads i, always after an occurrence of a $noncrit(i)$ action it holds that, as long as a $crit(i)$ action has not happened, a state is reachable in which $crit(i)$ is enabled.

The Reach property is implied by starvation freedom, and so if it does not hold, then neither does starvation freedom. We chose to analyse this property rather than starvation freedom itself because the presence of busy waiting loops in our models would require us to use fairness assumptions to dismiss spurious counterexamples. The question of how to interpret fairness assumptions when dealing with non-atomic registers is outside of the scope of this paper.

The results of our verification are shown in Table 1. When doing model checking, we have to instantiate a specific number of threads. We have restricted our verification to three threads for all algorithms, except for Dekker, Attiya-Welch and Peterson, which are only defined for two threads.

Table 1 Results of verifying mutual exclusion algorithms.

	Safe		Regular		Atomic	
	Mutex	Reach	Mutex	Reach	Mutex	Reach
Aravind (BLRU) [3, Figure 4]	✓	✓	✓	✓	✓	✓
Attiya-Welch [4, Algorithm 12]	✓	✓	✓	✓	✓	✓
Attiya-Welch alternate [29, Figure 19.1]	✓	×	✓	×	✓	✓
Dekker [1, Figure 1]	✓	✓	✓	✓	✓	✓
Dijkstra [9]	✓	✓	✓	✓	✓	✓
Knuth [17]	✓	✓	✓	✓	✓	✓
Lamport (3-bit) [20, Figure 2]	✓	✓	✓	✓	✓	✓
Peterson [27]	×	✓	×	✓	✓	✓
Szymanski (flag) [31, Figure 2]	×	×	×	✓	✓	✓
Szymanski (flag with bits)	×	✓	×	✓	×	✓
Szymanski (3-bit lin. wait) [32, Figure 1]	×	✓	×	✓	×	✓

In this section, we discuss some of our most interesting findings. For complete descriptions of counterexamples, as well as further discussion of our results we refer to [30, Appendix B]. All models are available through GitHub[5].

5.1 Peterson's Algorithm

Algorithm 1 Peterson's algorithm for two threads from [27]. We use i for the thread's own id and j for the other thread's id.

```
$flag[i] \leftarrow 1$
$turn \leftarrow i$
**await** $flag[j] = 0 \lor turn = j$
**critical section**
$flag[i] \leftarrow 0$
```

Peterson's classic algorithm (see Algorithm 1) was not designed to be correct under non-atomic register assumptions. An analysis of the mutual exclusion violation with safe registers still gives interesting insights into the algorithm and some of the unexpected behaviour of safe registers.

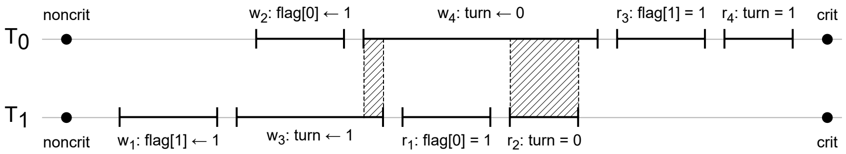


Figure 5 Counterexample generated by mCRL2 for mutual exclusion for Peterson's algorithm with safe registers, represented on a timeline.

[5] https://github.com/mCRL2org/mCRL2/tree/master/examples/academic/non-atomic_registers (972629b)

As expected, mCRL2 reports that mutual exclusion does not hold when using non-atomic registers. We present a visualisation of the counterexample generated by mCRL2 for safe registers in Figure 5. There are two instances of overlapping operations. First, since the two writes to *turn*, labelled w_3 and w_4 in Figure 5, overlap, according to the safe register model the register can have any arbitrary value after they both have ended. In this counterexample, *turn* has the value 1, which allows thread 0 to read the value 1 (the read labelled r_4) and enter the critical section. Second, thread 1's read of *turn* (labelled r_2) overlaps with thread 0's write (labelled w_4). The read can therefore return an arbitrary value, in this case the value 0, which allows thread 1 to enter the critical section.

This counterexample shows only overlaps on the *turn* register. We can initialise our model such that the *turn* register is atomic, but both *flag* registers behave as safe registers. We find that mutual exclusion does hold then. This confirms that overlapping operations on the *turn* register are the sole cause of the mutual exclusion violation for Peterson's algorithm. We discuss Peterson's algorithm with regular registers in [30, Appendix B].

5.2 Szymanski's Flag Algorithm

Algorithm 2 Szymanski's flag algorithm from [31], i is the thread's own id.

```
 1: $flag[i] \leftarrow 1$
 2: await $\forall j.\ flag[j] < 3$
 3: $flag[i] \leftarrow 3$
 4: if $\exists j.\ flag[j] = 1$ then
 5:     $flag[i] \leftarrow 2$
 6:     await $\exists j.\ flag[j] = 4$
 7: $flag[i] \leftarrow 4$
 8: await $\forall j < i.\ flag[j] < 2$
 9: critical section
10: await $\forall j > i.\ flag[j] < 2 \lor flag[j] > 3$
11: $flag[i] \leftarrow 0$
```

There are several variants of Szymanski's algorithm, which all seem to have been derived from the flag-based algorithm shown as Algorithm 2. In [31], Szymanski proposes this flag-based algorithm and claims that an implementation of it representing the flags using three bits is robust for flickering of bits (i.e., is correct for non-atomic registers). As indicated in Table 1, we find that neither the integer nor the bits variant ensure mutual exclusion when using non-atomic registers. The full analysis of the bits version, as well as a variant of it known as the 3-bit linear wait algorithm [32] are presented in [30, Appendix B]. Here, we only discuss the integer version of the flag algorithm, as the counterexample against Mutex that we have found illustrates the core issue shared by all mentioned variants of Szymanski's algorithm.

The pseudocode for the flag algorithm is shown in Algorithm 2. It is originally presented in [31, Figure 2], but note that we have repaired an obvious typo: [31, Figure 2] erroneously has a conjunction instead of a disjunction in line 10. All *flag* registers are initialised at 0.

See Figure 6 for a visualisation of the counterexample for mutual exclusion with two threads and regular registers that we found using the mCRL2 toolset. The first instance of a read overlapping with a write is irrelevant, reading $flag[1] = 1$ would also have been possible without overlap. The other two instances of overlap are of interest. Thread 0 is writing the value 3 to $flag[0]$ and thread 1 reads $flag[0]$ twice while this write is active. The first

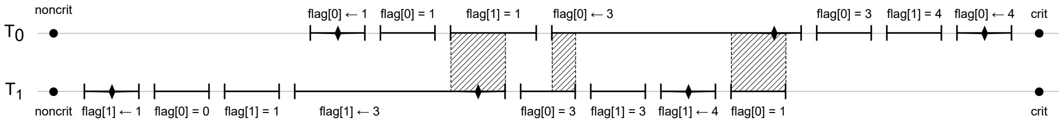


Figure 6 Mutex violation for Szymanski (flag) with regular registers and two threads, generated by mCRL2, on a timeline. The order-actions are drawn with lines during a write's execution.

time it reads the new value (3), while the second time it reads the old value (1). Lamport specifically highlights that such a sequence is possible when using regular registers [22]. Since only single-writer registers are used and write-order reduces to Lamport's definition of regular registers when single-writers in that case [29], this counterexample is also valid for write-order.

5.3 Implementation Details

Our analyses have also revealed that seemingly minor implementation subtleties can make the difference between a correct and an incorrect algorithm. A non-atomic register that is read multiple times in a row may return different values, even if no new writes to this register have started. This means that when the value of a register needs to be checked several times in an algorithm, there is a difference between reading it once and subsequently checking a local copy of the value, or reading it again when needed.

For an example where this affects correctness, consider the Attiya-Welch algorithm. While the presentation in [4, p. 77] ensures reachability of the critical section with safe registers, the seemingly equivalent reformulation of this same algorithm in [29] does not. The latter suggests that a thread needs to read a particular register twice as part of two different conditions that in the former are handled simultaneously. In [29], that presentation of the algorithm is claimed to be correct under all four of their MWMR regular register models; our counterexample shows that it is not. A similar phenomenon occurs with Lamport's 3-bit algorithm, in which each thread i has a bit z_i. As part of the algorithm, a computation is done on z (the function assigning z_i to i). Lamport states that "evaluating $[z]$ at j requires a read of the variable z_j." This may lead one to implement this algorithm by having threads re-read variables whenever needed. It turns out this implementation leads to a deadlock. Locally saving all required z-values at the start of the computation and then only referencing this local copy during the computation solves this issue. Consequently, these algorithms have a correct implementation, but they are also easily implemented incorrectly. See the discussions of Attiya-Welch and Lamport in [30, Appendix B] for more details.

5.4 Other Verifications

There have been many mechanical verifications of mutual exclusion algorithms with atomic registers. For instance, in recent tutorials on the verification of distributed algorithms in mCRL2, verifications of Dekker's and Peterson's algorithms are presented [12, 13]. Several such verifications have also been done with the CADP toolset; see, e.g., [26] for the results of verifying a large number of mutual exclusion algorithms, including Szymanski, Dekker and Peterson, with atomic registers.

To the best of our knowledge, we are the first to propose a systematic approach to mechanically verifying the correctness of mutual exclusion algorithms with respect to non-atomic registers, but there have been some mechanical verifications for specific algorithms.

Lamport himself modelled the Bakery algorithm in TLA+, representing the non-atomic writes as sequences of write actions of arbitrary length, where every action results in an arbitrary value being written, except for the last which writes the intended value [23]. This approach for modelling safe registers only works for SWMR registers; it does not work for MWMR registers. This approach for modelling safe SWMR registers, as well as a similar approach for modelling regular SWMR registers, is presented in [2]. This approach is also used in several verifications done by Wim Hesselink, including of the Lycklama–Hadzilacos–Aravind algorithm in [16] and the Bakery algorithm in [15].

In [8], several mutual exclusion algorithms are verified with atomic registers using timed automata in UPPAAL. Additionally, the Block & Woo algorithm is checked with bit flickering. Their model does not account for writes that overlap with other writes. Additionally, their model for the behaviour of safe registers is specific to the registers used in the algorithm.

Dekker's algorithm with safe registers is considered in [6]. There it is demonstrated that Dekker's algorithm does not satisfy starvation freedom when safe registers are used, and a fixed version of the algorithm is presented.

Szymanski's flag algorithm with atomic registers is proven correct in [25]. This paper demonstrates the importance of checking all threads in the "forall" and "exists" statements in the pseudocode in the same order every time. This is also how we model the algorithm.

There have been other verifications of Szymanski's algorithms [24, 33], the former paper using the STeP tool. However, the exact pseudocode in those proofs differs from the pseudocode in [31] and [32].

6 Conclusions

We have presented process-algebraic models of safe, regular and atomic multi-writer multi-reader registers and used them to determine the robustness of various mutual exclusion algorithms for relaxed atomicity assumptions. Our analyses revealed issues with several of the algorithms discussed.

There are many more mutual exclusion algorithms that could be analysed in the same way as the ones shown in Section 5. In [32], Szymanski presents three other mutual exclusion algorithms. There also exist several variants of Szymanski's algorithm [24, 33], all of which are similar to the 3-bit linear wait algorithm but differ in small ways. In [6] it is shown that Dekker's algorithm does not ensure starvation freedom when safe registers are used and a modified version of the algorithm is presented which does satisfy this property. When we add verification of starvation freedom to our analysis, we can confirm their work.

We have only considered to what extent various algorithms guarantee mutual exclusion and whether the critical section is always reachable for every thread. Our next step will be to consider starvation freedom. Van Glabbeek proves that starvation freedom cannot hold for any mutual exclusion algorithm for which the correctness, on the one hand, relies on atomicity of memory interactions and, on the other hand, does not rely on assumptions regarding the relative speeds of threads [10]. A crucial presupposition for his argument is that a convincing verification hinges on not more than a component-based fairness assumption called justness [11]. In [5] a method is proposed for verifying liveness properties under justness assumptions using the mCRL2 toolset. The method requires a classification of the roles of components in interactions. It should be investigated how to classify the roles of threads and registers in invocations and responses, and, in particular, how to deal with the ow_i, ew_i and er_i actions.

References

1 K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *ACM SIGACT News*, 34(3):94–103, 2003.

2 James H. Anderson and Mohamed G. Gouda. Atomic semantics of nonatomic programs. *Information Processing Letters*, 28(2):99–103, 1988.

3 Alex A. Aravind. Yet another simple solution for the concurrent programming control problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1056–1063, 2010.

4 Hagit Attiya and Jennifer L. Welch. *Distributed computing – Fundamentals, simulations, and advanced topics (2. ed.).* Wiley series on parallel and distributed computing. Wiley, 2004.

5 Mark Bouwman, Bas Luttik, and Tim A. C. Willemse. Off-the-shelf automated analysis of liveness properties for just paths. *Acta Informatica*, 57(3-5):551–590, 2020. doi:10.1007/s00236-020-00371-w.

6 Peter A. Buhr, David Dice, and Wim H. Hesselink. Dekker's mutual exclusion algorithm made rw-safe. *Concurrency and Computation: Practice and Experience*, 28(1):144–165, 2016.

7 Olav Bunte, Jan Friso Groote, Jeroen J.A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A.C. Willemse. The mCRL2 toolset for analysing concurrent systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39. Springer, 2019.

8 Franco Cicirelli and Libero Nigro. Modelling and verification of mutual exclusion algorithms. In *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 136–144. IEEE, 2016.

9 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

10 Rob van Glabbeek. Modelling mutual exclusion in a process algebra with time-outs. *CoRR*, abs/2106.12785, 2021. arXiv:2106.12785.

11 Rob van Glabbeek and Peter Höfner. Progress, Justness, and Fairness. *ACM Computing Surveys (CSUR)*, 52(4):1–38, 2019.

12 Jan Friso Groote and Jeroen J. A. Keiren. Tutorial: designing distributed software in mCRL2. In *Formal Techniques for Distributed Objects, Components, and Systems: 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14–18, 2021, Proceedings*, pages 226–243. Springer, 2021.

13 Jan Friso Groote, Jeroen J. A. Keiren, Bas Luttik, Erik P. de Vink, and Tim A. C. Willemse. Modelling and analysing software in mCRL2. In Farhad Arbab and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software – 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings*, volume 12018 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2019. doi:10.1007/978-3-030-40914-2_2.

14 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems.* MIT Press Ltd., 2014.

15 Wim H. Hesselink. Mechanical verification of Lamport's bakery algorithm. *Science of Computer Programming*, 78(9):1622–1638, 2013.

16 Wim H. Hesselink. Mutual exclusion by four shared bits with not more than quadratic complexity. *Science of Computer Programming*, 102:57–75, 2015.

17 Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

18 Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974. doi:10.1145/361082.361093.

19 Leslie Lamport. The mutual exclusion problem: Part I – A theory of interprocess communication. *J. ACM*, 33(2):313–326, April 1986. doi:10.1145/5383.5384.

20 Leslie Lamport. The mutual exclusion problem: Part II – Statement and solutions. *J. ACM*, 33(2):327–348, April 1986. doi:10.1145/5383.5385.

21 Leslie Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Comput.*, 1(2):77–85, 1986. doi:10.1007/BF01786227.

22 Leslie Lamport. On interprocess communication. Part II: Algorithms. *Distributed Comput.*, 1(2):86–101, 1986. doi:10.1007/BF01786228.

23 Leslie Lamport. The TLA+ Hyperbook, August 2015. Available at http://lamport.azurewebsites.net/tla/hyperbook.html, accessed on 26 April 2023, see Chapter 7.8.4.

24 Zohar Manna, Anuchit Anuchitanukul, Nikolaj Bjorner, Anca Browne, and Edward Chang. STeP: The Stanford Temporal Prover. Technical report, Stanford University Department of Computer Science, 1994.

25 Zohar Manna and Amir Pnueli. An exercise in the verification of multi-process programs. *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 289–301, 1990.

26 Radu Mateescu and Wendelin Serwe. A study of shared-memory mutual exclusion protocols using CADP. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 180–197. Springer, 2010.

27 Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981. doi:10.1016/0020-0190(81)90106-X.

28 Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2013. doi:10.1007/978-3-642-32027-9.

29 Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011. doi:10.1137/07071158X.

30 Myrthe Spronck and Bas Luttik. Process-algebraic models of multi-writer multi-reader non-atomic registers, 2023. arXiv:2307.05143.

31 Boleslaw K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In Jacques Lenfant, editor, *Proceedings of the 2nd international conference on Supercomputing, ICS 1988, Saint Malo, France, July 4-8, 1988*, pages 621–626. ACM, 1988. doi:10.1145/55364.55425.

32 Boleslaw K. Szymanski. Mutual exclusion revisited. In Joshua Maor and Abraham Peled, editors, *Next Decade in Information Technology: Proceedings of the 5th Jerusalem Conference on Information Technology 1990, Jerusalem, October 22-25, 1990*, pages 110–117. IEEE Computer Society, 1990. doi:10.1109/JCIT.1990.128275.

33 Boleslaw K. Szymanski and Jose M. Vidal. Automatic verification of a class of symmetric parallel programs. In *IFIP Congress (1)*, pages 571–576, 1994.

Geometry of Reachability Sets of Vector Addition Systems

Roland Guttenberg
Technical University of Munich, Germany

Mikhail Raskin
LaBRI, University of Bordeaux, France

Javier Esparza
Technical University of Munich, Germany

Abstract

Vector Addition Systems (VAS), aka Petri nets, are a popular model of concurrency. The reachability set of a VAS is the set of configurations reachable from the initial configuration. Leroux has studied the geometric properties of VAS reachability sets, and used them to derive decision procedures for important analysis problems. In this paper we continue the geometric study of reachability sets. We show that every reachability set admits a finite decomposition into disjoint almost hybridlinear sets enjoying nice geometric properties. Further, we prove that the decomposition of the reachability set of a given VAS is effectively computable. As a corollary, we derive a new proof of Hauschildt's 1990 result showing the decidability of the question whether the reachability set of a given VAS is semilinear. As a second corollary, we prove that the complement of a reachability set, if it is infinite, always contains an infinite linear set.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Vector Addition System, Petri net, Reachability Set, Almost hybridlinear, Partition, Geometry

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.6

Related Version *Full Version*: `https://arxiv.org/abs/2211.02889` [5]

Funding The project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367.

1 Introduction

Vector Addition Systems (VAS), also known as Petri nets, are a popular model of concurrent systems. The VAS reachability problem consists of deciding if a target configuration of a VAS is reachable from some initial configuration. It was proved decidable in the 1980s [8,17], but its complexity (Ackermann-complete) could only be determined recently [2,3,14].

The *reachability set* of a VAS is the set of all configurations reachable from the initial configuration. Configurations are tuples of natural numbers, and so the reachability set of a VAS is a subset of $\mathbb{N}^n$ for some n called the *dimension* of the VAS. Results on the geometric properties of reachability sets have led to new algorithms in the past. For example, in [12] it was shown that every configuration outside the reachability set $\mathbf{R}$ of a VAS is separated from $\mathbf{R}$ by a semilinear inductive invariant. This immediately leads to an algorithm for the reachability problem consisting of two semi-algorithms, one enumerating all possible paths to certify reachability, and one enumerating all semilinear sets and checking if they are separating inductive invariants. Another example is [13], where it was shown that semilinear reachability sets are flatable. The result led to an algorithm for deciding whether a semilinear set is included in or equal to the reachability set of a given VAS.

The separability and flatability results of [12,13] are proven not only for VAS reachability sets, but for arbitrary semilinear *Petri sets*, a larger class with a geometric definition introduced in [12]. So, in particular, [13] is an investigation into the geometric structure of semilinear Petri sets. In this paper we study the structure of the *non-semilinear* Petri sets. We introduce hybridization, or, equivalently, the class of *almost hybridlinear* sets, a generalization of the hybridlinear sets introduced by Ginsburg and Spanier [4] and further studied by Chistikov and Haase [1]. We prove the following decomposition:

▶ **Theorem 1.1.** *Let $\mathbf{X}$ be a Petri set. For every semilinear set $\mathbf{S}$ there exists a partition $\mathbf{S} = \mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ into pairwise disjoint full linear sets such that for all $i \in \{1, \ldots, k\}$ either $\mathbf{X} \cap \mathbf{S}_i = \emptyset$, $\mathbf{S}_i \subseteq \mathbf{X}$ or $\mathbf{X} \cap \mathbf{S}_i$ is irreducible with hybridization $\mathbf{S}_i$. Further, if $\mathbf{X}$ is the reachability set of a VAS, then the partition is computable.*

Defining hybridization and irreducibility is beyond the scope of this introduction; in fact, they will be introduced in Section 4 and 5 of this paper. However, we can already explain two properties of the irreducible sets with a hybridization which, combined with Theorem 1.1, have important consequences.

Firstly, irreducible sets with hybridization are always non-semilinear. This leads to a simple algorithm for deciding whether the reachability set $\mathbf{X} \subseteq \mathbb{N}^d$ of a given VAS of dimension d is semilinear. Let $\mathbf{S} := \mathbb{N}^d$ and compute the partition $\mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ of Theorem 1.1. For every $1 \leq i \leq k$, check whether $\mathbf{X} \cap \mathbf{S}_i = \emptyset$ or $\mathbf{S}_i \subseteq \mathbf{X}$ hold[1]. If this is the case for all i, then let J be the set of indices i, where $\mathbf{S}_i \subseteq \mathbf{X}$ holds. We have $\bigcup_{i \in J} \mathbf{S}_i = \mathbf{X} \cap \mathbf{S} = \mathbf{X}$, and so, since $\mathbf{S}_1, \ldots, \mathbf{S}_k$ are linear, $\mathbf{X}$ is semilinear. Otherwise, by Theorem 1.1 there exists an i such that $\mathbf{X} \cap \mathbf{S}_i$ is irreducible with hybridization $\mathbf{S}_i$, and hence non-semilinear. Since semilinear sets are closed under intersection, $\mathbf{X}$ is not semilinear. The decidability of the semilinearity of VAS reachability sets was first proved by Hauschildt [6], and in fact we arrive at essentially the same algorithm. However, we provide a simpler correctness proof and a clear geometric intuition. Further, our theorem holds for arbitrary Petri sets, a larger class than VAS reachability sets.

Secondly, if a set $\mathbf{X}$ is irreducible with hybridization $\mathbf{S}$, then there are infinitely many points in the boundary $\partial \mathbf{S}$ of $\mathbf{S}$ that do not belong to $\mathbf{S}$, i.e., $|\partial \mathbf{S} \setminus \mathbf{X}| = \infty$. This allows to prove that if $\mathbf{S} \setminus \mathbf{X}$ is infinite, then $\mathbf{S} \setminus \mathbf{X}$ contains an infinite linear set, which was left as a conjecture in [7]. Namely the proof is now a simple induction on the dimension of the semilinear set $\mathbf{S}$: If $\mathbf{S} \setminus \mathbf{X}$ is infinite, then some $\mathbf{S}_i \setminus \mathbf{X}$ is infinite. If for this i, we have $\mathbf{X} \cap \mathbf{S}_i = \emptyset$ or $\mathbf{S}_i \subseteq \mathbf{X}$, then $\mathbf{S}_i \setminus \mathbf{X}$ is semilinear and hence contains an infinite line. Otherwise we have that $|\partial \mathbf{S}_i \setminus \mathbf{X}| = \infty$, and hence by induction $\partial \mathbf{S}_i \setminus \mathbf{X}$ contains an infinite line. This corollary is a first step towards understanding the complements of VAS reachability sets, for which little is known.

The sections of the paper follow the structure of the main theorem. Section 2 contains preliminaries. Section 3 introduces smooth sets, preparing for the introduction of hybridization and Petri sets in Section 4. Section 5 introduces irreducibility and proves Theorem 1.1. Section 6 proves the corollaries of Theorem 1.1.

2 Preliminaries

We let $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{Q}_{\geq 0}$ denote the natural, integer, and (non-negative) rational numbers.

[1] It is well known that the first question can be reduced to the VAS reachability problem, and the second is decidable by the flatability results mentioned before.

Furthermore, we use uppercase letters except A for sets, with A being used for matrices. We use boldface for vectors and sets of vectors. We denote the cardinality of a set $\mathbf{X}$ as $|\mathbf{X}|$.

Given sets $\mathbf{X}, \mathbf{Y} \subseteq \mathbb{Q}^n, Z \subseteq \mathbb{Q}$, we write $\mathbf{X} + \mathbf{Y} := \{\mathbf{x} + \mathbf{y} \mid \mathbf{x} \in \mathbf{X}, \mathbf{y} \in \mathbf{Y}\}$ and $Z \cdot \mathbf{X} := \{\lambda \cdot \mathbf{x} \mid \lambda \in Z, \mathbf{x} \in \mathbf{X}\}$. By identifying elements $\mathbf{x} \in \mathbb{Q}^n$ with $\{\mathbf{x}\}$, we define $\mathbf{x} + \mathbf{X} := \{\mathbf{x}\} + \mathbf{X}$, and similarly $\lambda \cdot \mathbf{X} := \{\lambda\} \cdot \mathbf{X}$ for $\lambda \in \mathbb{Q}$. We denote by $\mathbf{X}^C$ the complement of $\mathbf{X}$. On $\mathbb{Q}^n$, we consider the usual Euclidean norm and its generated topology. We denote the closure of a set $\mathbf{X}$ in this topology by $\overline{\mathbf{X}}$.

A *vector space* $\mathbf{V} \subseteq \mathbb{Q}^n$ is a set such that $\mathbf{0} \in \mathbf{V}$, $\mathbf{V} + \mathbf{V} \subseteq \mathbf{V}$ and $\mathbb{Q} \cdot \mathbf{V} \subseteq \mathbf{V}$. Given a set $\mathbf{F} \subseteq \mathbb{Q}^n$, the vector space generated by $\mathbf{F}$ is the smallest vector space containing $\mathbf{F}$. Every vector space $\mathbf{V}$ is *finitely generated* (f.g.), i.e. there exists a finite set $\mathbf{F} \subseteq \mathbb{Q}^n$ generating $\mathbf{V}$. Furthermore, it can also be expressed as $\{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} = 0\}$ for some integer matrix A.

2.1 Cones, lattices, and periodic sets

A set $\mathbf{C} \subseteq \mathbb{Q}^n$ is a *cone* if $\mathbf{0} \in \mathbf{C}$, $\mathbf{C} + \mathbf{C} \subseteq \mathbf{C}$ and $\mathbb{Q}_{>0}\mathbf{C} \subseteq \mathbf{C}$. Given a set $\mathbf{F} \subseteq \mathbb{Q}^n$, the cone generated by $\mathbf{F}$ is the smallest cone containing $\mathbf{F}$. If $\mathbf{C}$ is a cone, then $\mathbf{C} - \mathbf{C}$ is the vector space generated by $\mathbf{C}$. Not every cone is finitely generated (f.g.). Instead, we have:

▶ **Lemma 2.1** ([19, Corollary 7.1a]). *Let $\mathbf{C} \subseteq \mathbb{Q}^n$ be a cone. Then $\mathbf{C}$ is finitely generated if and only if $\mathbf{C} = \{\mathbf{x} \in \mathbf{C} - \mathbf{C} \mid A\mathbf{x} \geq \mathbf{0}\}$ for some integer matrix A.*

In particular, finitely generated cones are closed. The *interior* of a finitely generated cone $\mathbf{C}$ is the set $\text{int}(\mathbf{C}) = \{\mathbf{x} \in \mathbf{C} - \mathbf{C} \mid A\mathbf{x} > \mathbf{0}\}$, where A is a matrix as above. The boundary of the cone is $\partial(\mathbf{C}) := \overline{\mathbf{C}} \setminus \text{int}(\mathbf{C})$. It is well known that the boundary of a cone is a a finite union of lower dimensional cones, called facets [19]. In fact, there is a defining matrix A such that the facets are exactly the sets of solutions obtained by changing one of the inequalities of $A\mathbf{x} \geq 0$ into an equality. For example, the left part of Figure 1 shows the cone $\{(x, y) \mid x - y \geq y, y \geq 0\}$. Its facets are the sets $\{(x, y) \mid x - y = 0, y \geq 0\}$ and $\{(x, y) \mid x \geq y, y = 0\}$ (shown as black lines in the picture), and their union is the boundary of the cone.

A cone $\mathbf{C}$ is *definable* if it is definable in $\text{FO}(\mathbb{Q}, +, \geq)$. A cone $\mathbf{C}$ is definable iff $\mathbf{C} \setminus \{\mathbf{0}\} = \{\mathbf{x} \in \mathbf{C} - \mathbf{C} \mid A_1\mathbf{x} > \mathbf{0}, A_2\mathbf{x} \geq \mathbf{0}\}$ for some integer matrices A_1, A_2. In this case the closure $\overline{\mathbf{C}}$ is finitely generated. Intuitively, changing an equation from from ≥ 0 to > 0 removes a facet. Removing all facets yields $\text{int}(\mathbf{C})$.

A set $\mathbf{L} \subseteq \mathbb{Z}^n$ is a *lattice* if $\mathbf{L} + \mathbf{L} \subseteq \mathbf{L}, -\mathbf{L} \subseteq \mathbf{L}$ and $\mathbf{0} \in \mathbf{L}$. For any finite set $\mathbf{F} = \{\mathbf{x}_1, \ldots, \mathbf{x}_s\} \subseteq \mathbb{N}^n$, the lattice generated by $\mathbf{F}$ is $\mathbb{Z}\mathbf{x}_1 + \cdots + \mathbb{Z}\mathbf{x}_s$. Every lattice is finitely generated, and even has a generating set linearly independent over $\mathbb{Q}$.

A set $\mathbf{P} \subseteq \mathbb{N}^n$ is a *periodic set* if $\mathbf{P} + \mathbf{P} \subseteq \mathbf{P}$ and $\mathbf{0} \in \mathbf{P}$. For any set $\mathbf{F} \subseteq \mathbb{N}^n$, the periodic set $\mathbf{F}^*$ generated by $\mathbf{F}$ is the smallest periodic set containing $\mathbf{F}$. We have $\mathbf{F}^* = \{\mathbf{p}_1 + \cdots + \mathbf{p}_r \mid r \in \mathbb{N}, \mathbf{p}_i \in \mathbf{F} \text{ for all } i\}$. A periodic set $\mathbf{P}$ is *finitely generated* if $\mathbf{P} = \mathbf{F}^*$ for some finite set $\mathbf{F}$. Finitely generated periodic sets are characterized as follows:

▶ **Lemma 2.2** ([13, Lemma V.5]). *Let $\mathbf{P} \subseteq \mathbb{N}^n$ be a periodic set. Then $\mathbf{P}$ is finitely generated as a periodic set if and only if $\mathbb{Q}_{\geq 0}\mathbf{P}$ is finitely generated as a cone.*

Any set generates a lattice, a cone and a vector space. In the case of periodic sets these have simple formulas; namely $\mathbf{P} - \mathbf{P}$, as well as $\mathbb{Q}_{\geq 0}\mathbf{P}$ and $\text{VectSp}(\mathbf{P}) := \mathbb{Q}_{\geq 0}(\mathbf{P} - \mathbf{P}) = \mathbb{Q}_{\geq 0}\mathbf{P} - \mathbb{Q}_{\geq 0}\mathbf{P}$ respectively. These are also depicted in the right of Figure 1. On the other hand, if $\mathbf{C}$ is a cone and $\mathbf{L}$ is a lattice, then $\mathbf{C} \cap \mathbf{L}$ is a periodic set. We will consider periodic sets of this form in more depth in Section 2.3.

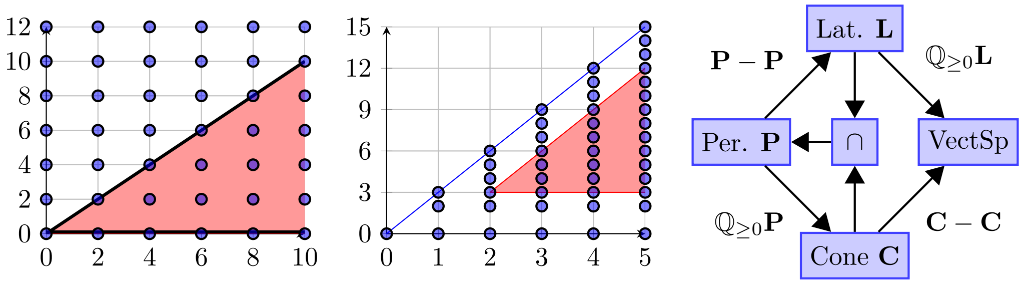


Figure 1 *Left*: The cone generated by $\{(1,1),(1,0)\}$ is shown in red, with its boundary in black. The lattice $(2,0)\mathbb{Z}+(0,2)\mathbb{Z}$ is the set of of blue dots. Their intersection is the periodic set $\{(2,0),(2,2)\}^*$.
Middle: The periodic set $\mathbf{P}=\{(1,0),(1,2),(1,3)\}^*$ is shown in blue. Intuitively, the set $\{(1,1),(2,1),(3,1),\ldots\}$ is a "hole" of $\mathbf{P}$. Inside $\mathbf{P}$ we find the red area $(2,3)+\mathbf{P}$, whose blue points do not intersect the hole, i.e., $(2,3)+\text{Fill}(\mathbf{P})\subseteq\mathbf{P}$.
Right: Graph comparing the classes of sets defined in Section 2.

2.2 Dimension

The *dimension* of a vector space defined as its number of generators is a well-known concept. It can be extended to arbitrary subsets of $\mathbb{Q}^n$ as follows.

▶ **Definition 2.3** ([11, 12]). *Let $\mathbf{X}\subseteq\mathbb{Q}^n$. The* dimension *of $\mathbf{X}$, denoted $\dim(\mathbf{X})$, is the smallest natural number k such that there exist finitely many vector spaces $\mathbf{V}_i\subseteq\mathbb{Q}^n$ with $\dim(\mathbf{V}_i)\leq k$ and vectors $\mathbf{b}_i\in\mathbb{Q}^n$ such that $\mathbf{X}\subseteq\bigcup_{i=1}^r\mathbf{b}_i+\mathbf{V}_i$.*

This dimension function has the following properties.

▶ **Lemma 2.4.** *Let $\mathbf{X},\mathbf{X}'\subseteq\mathbb{Q}^n,\mathbf{b}\in\mathbb{Q}^n$. Then $\dim(\mathbf{X})=\dim(\mathbf{b}+\mathbf{X})$ and $\dim(\mathbf{X}\cup\mathbf{X}')=\max\{\dim(\mathbf{X}),\dim(\mathbf{X}')\}$. Further, if $\mathbf{X}\subseteq\mathbf{X}'$, then $\dim(\mathbf{X})\leq\dim(\mathbf{X}')$.*

▶ **Lemma 2.5** ([11, Lemma 5.3]). *Let $\mathbf{P}$ be periodic. Then $\dim(\mathbf{P})=\dim(\text{VectSp}(\mathbf{P}))$.*

Lemma 2.5 for example shows that the lattice and the cone depicted in the left of Figure 1, as well as the periodic set obtained as intersection have dimension 2, because all of them generate the vector space $\mathbb{Q}^2$.

2.3 Finitely generated vs. full periodic sets

A set $\mathbf{L}$ is *linear* if $\mathbf{L}=\mathbf{b}+\mathbf{P}$ with $\mathbf{b}\in\mathbb{N}^n$ and $\mathbf{P}\subseteq\mathbb{N}^n$ a finitely generated periodic set. A set $\mathbf{S}$ is *semilinear* if it is a finite union of linear sets. The semilinear sets coincide with the sets definable via formulas $\varphi\in\text{FO}(\mathbb{N},+,\geq)$, also called Presburger Arithmetic. This is the usual definition of a linear set in theoretical computer science, however, we will work with a slightly smaller class of linear sets, which we call full linear sets. As shown for example in [20], working with this smaller class does not change the class of semilinear sets: A set $\mathbf{S}$ is semilinear if and only if it is a finite union of full linear sets, i.e. linear sets $\mathbf{b}+\mathbf{P}$ where $\mathbf{P}$ is not only finitely generated, but even full, as in the following definition.

▶ **Definition 2.6.** *A periodic $\mathbf{P}$ is* full *if $\mathbf{P}=\mathbf{C}\cap\mathbf{L}$, where $\mathbf{C}$ is a f.g. cone and $\mathbf{L}$ a lattice.*

Full linear sets have even been used as the main definition of linear set in the literature before, for example in [18]. Furthermore, while not directly defined, this class was also utilized in [12, 13] as well. For an example of a finitely generated periodic set which is not full, consider the middle of Figure 1.

There is another equivalent definition of full periodic sets, which uses an overapproximation of a periodic set we call $\text{Fill}(\mathbf{P})$. This overapproximation was first introduced in [11] with the terminology $\text{lin}(\mathbf{P})$. However, we avoid this terminology because in [12, 13], the same author used the same notation with a slightly different meaning.

▶ **Definition 2.7.** *Let $\mathbf{P}$ be a periodic set. The* fill *of $\mathbf{P}$ is the set* $\text{Fill}(\mathbf{P}) := (\mathbf{P} - \mathbf{P}) \cap \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$.

Intuitively, we overapproximate $\mathbf{P}$ via the intersection of the obvious lattice and cone. The reason for using the closure of $\mathbb{Q}_{\geq 0}\mathbf{P}$ instead of the cone $\mathbb{Q}_{\geq 0}\mathbf{P}$ itself is Lemma 2.2: If the cone is not closed, then the periodic set, in our case $\text{Fill}(\mathbf{P})$, is not finitely generated. If $\mathbf{P}$ was already finitely generated, the definitions coincide.

▶ **Lemma 2.8.** *A periodic set $\mathbf{P}$ is full if and only if $\overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$ is a f.g. cone and $\mathbf{P} = \text{Fill}(\mathbf{P})$.*

By Lemma 2.2, full periodic sets are finitely generated: Namely, their cone $\mathbb{Q}_{\geq 0}\mathbf{P}$ equals $\mathbf{C} \cap \mathbb{Q}_{\geq 0}\mathbf{L}$, which as intersection of f.g. cones is finitely generated by Lemma 2.1.

Let us conclude this subsection with the main advantage of full linear over linear sets.

▶ **Lemma 2.9.** *Let $\mathbf{P}, \mathbf{Q}$ periodic, $\mathbf{P}$ full, $\mathbf{b}, \mathbf{c} \in \mathbb{Q}^n$ such that $\mathbf{c} + \mathbf{Q} \subseteq \mathbf{b} + \mathbf{P}$. Then $\mathbf{Q} \subseteq \mathbf{P}$.*

Proof. Since $\mathbf{P}$ is full, by Lemma 2.8 it is sufficient to prove $\mathbf{Q} \subseteq \mathbf{P} - \mathbf{P}$ and $\mathbf{Q} \subseteq \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$.

To prove $\mathbf{Q} \subseteq \mathbf{P} - \mathbf{P}$, observe that $\mathbf{Q} = (\mathbf{c} + \mathbf{Q}) - \mathbf{c} \subseteq (\mathbf{b} + \mathbf{P}) - (\mathbf{b} + \mathbf{P}) = \mathbf{P} - \mathbf{P}$.

To prove $\mathbf{Q} \subseteq \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$, write $\overline{\mathbb{Q}_{\geq 0}\mathbf{P}} = \{\mathbf{x} \in \text{VectSp}(\mathbf{P}) \mid A\mathbf{x} \geq 0\}$ for a matrix A, as in Lemma 2.1. Let A_k be the k-th row of A. It suffices to show $A_k\mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbf{Q}$. If we had $A_k\mathbf{x} < 0$, then $A_k(\mathbf{c} + \lambda\mathbf{x}) < A_k\mathbf{b}$ for large enough λ, contradicting $\mathbf{c} + \mathbf{Q} \subseteq \mathbf{b} + \mathbf{P}$. ◀

Observe that if we replace full by finitely generated, then the lemma does not hold: Choose $\mathbf{P}$ as the periodic set in the middle of Figure 1, then $(2, 3) + \{(1, 1)\}^* \subseteq \mathbf{P}$, and the property is violated, since $(1, 1) \notin \mathbf{P}$.

Another advantage is that many proofs simplify in the full case. The following such case will be a cornerstone of our main algorithm:

▶ **Lemma 2.10** ([13, Corollary D.3])**.** *Let $\mathbf{P}$ be a finitely generated periodic set. For every $\mathbf{x} \in \mathbf{P}$ the set $\mathbf{S} := \mathbf{P} \setminus (\mathbf{x} + \mathbf{P})$ is semilinear and satisfies $\dim(\mathbf{S}) < \dim(\mathbf{P})$.*

To prove this, first show that $\mathbf{P}$ contains $\mathbf{v} + \text{Fill}(\mathbf{P})$, as in the middle of Figure 1, and reduce to the case of full periodic $\mathbf{P}$. For full $\mathbf{P}$ it is geometrically clear; for example removing the red cone in the middle of Figure 1 from the set, we are left with a finite union of lines.

3 Smooth Periodic Sets

Not all periodic sets we need in the paper are finitely generated, but they are smooth, a class introduced by Leroux in [13]. Intuitively, a smooth set $\mathbf{P}$ is "close" to being finitely generated, in the sense that $\text{Fill}(\mathbf{P})$ is finitely generated. This result (very similar to a result of [11]) is proven in Section 3.1. In the rest of the section we show that smooth sets satisfying a novel condition are closed under intersection and enjoy good properties (Proposition 3.9).

We first reintroduce the set of directions of a periodic set.

▶ **Definition 3.1** ([13])**.** *Let $\mathbf{P}$ be a periodic set. A vector $\mathbf{d} \in \mathbb{Q}^n$ is a* direction *of $\mathbf{P}$ if there exists $m \in \mathbb{N}_{>0}$ and a point $\mathbf{x}$ such that $\mathbf{x} + \mathbb{N} \cdot m\mathbf{d} \subseteq \mathbf{P}$, i.e. some line in direction $\mathbf{d}$ is fully contained in $\mathbf{P}$. The set of directions of $\mathbf{P}$ is denoted* $\text{dir}(\mathbf{P})$.

We can now define smooth periodic sets.

▶ **Definition 3.2** ([13]). *Let* $\mathbf{P}$ *be a periodic set.*

- $\mathbf{P}$ *is* asymptotically definable *if* $\mathrm{dir}(\mathbf{P})$ *is a definable cone, i.e.* $\mathrm{dir}(\mathbf{P}) \setminus \{\mathbf{0}\} = \{\mathbf{x} \in \mathrm{VectSp}(\mathbf{P}) \mid A_1\mathbf{x} > 0, A_2\mathbf{x} \geq 0\}$ *for some integer matrices* A_1, A_2*.*
- $\mathbf{P}$ *is* well-directed *if every sequence* $(\mathbf{p}_m)_{m\in\mathbb{N}}$ *of vectors* $\mathbf{p}_m \in \mathbf{P}$ *has an infinite subsequence* $(\mathbf{p}_{m_k})_{k\in\mathbb{N}}$ *such that* $\mathbf{p}_{m_k} - \mathbf{p}_{m_j} \in \mathrm{dir}(\mathbf{P})$ *for all* $k \geq j$*.*
- $\mathbf{P}$ *is* smooth *if it is asymptotically definable and well-directed.*

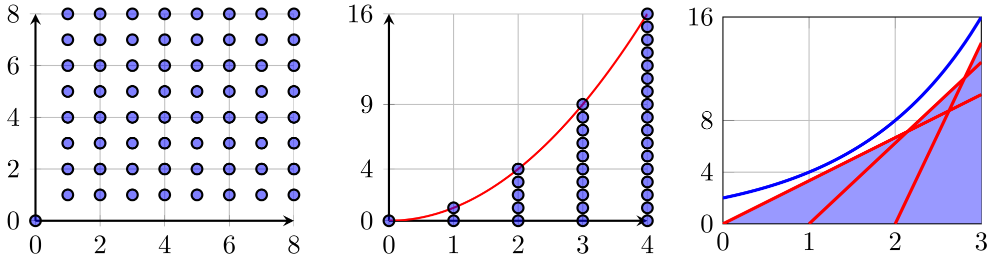


Figure 2 *Left and middle*: The periodic sets $\mathbf{P} = \{(0,0)\} \cup \mathbb{N}^2_{>0}$ and $\mathbf{P} = \{(x,y) \in \mathbb{N}^2 \mid y \leq x^2\}$ respectively. Neither is finitely generated, but both are smooth with $\mathrm{Fill}(\mathbf{P}) = \mathbb{N}^2$.
Right: Underapproximation of $\{(x,y) \mid y \leq 2^{x+1}\}$ via a union of three cones. The starting points are respectively $(0,0), (1,0)$ and $(2,0)$.

Figure 2 shows two examples of smooth periodic sets that are not finitely generated.

▶ **Example 3.3.** Examples of non-smooth sets are $\mathbf{P}_1 = \{(x,y) \mid x \geq \sqrt{2}y\}$ and $\mathbf{P}_2 = (\{(0,1)\} \cup \{(2^m,1) \mid m \in \mathbb{N}\})^* = \{(x,n) \in \mathbb{N}^2 \mid x$ has at most n bits set to 1 in the binary representation.$\}$. $\mathbf{P}_1$ is not asymptotically definable, because defining $\mathrm{dir}(\mathbf{P})$ requires irrationals, while $\mathbf{P}_2$ is not well-directed (see observation 2 below).

Intuitively, the “boundaries” of a smooth periodic set in two dimensions are either straight lines or function graphs “curving outward”, as in the example on the right of Figure 2.

We make a few observations:

1. The set $\mathrm{dir}(\mathbf{P})$ is a cone. Indeed, if two lines in different directions $\mathbf{d}$ and $\mathbf{d}'$ are contained in $\mathbf{P}$, then by periodicity $\mathbf{P}$ also contains a $\mathbf{d}, \mathbf{d}'$ plane, and so $\mathbf{P}$ contains a line in every direction between $\mathbf{d}$ and $\mathbf{d}'$.
2. The most important case of Definition 3.2 is when the $\mathbf{p}_m$ are all on the same infinite line $\mathbf{x} + \mathbf{d} \cdot \mathbb{N}$. Then the definition equivalently states that $\mathbf{d} \in \mathrm{dir}(\mathbf{P})$, i.e. some infinite line in direction $\mathbf{d}$ is contained in $\mathbf{P}$. This makes sets where points are “too scarce” non-smooth. For instance, the set $\mathbf{P}_2$ of Example 3.3 contains infinitely many points on a horizontal line, but no full horizontal line, which would correspond to an arithmetic progression.

3.1 Fills of Smooth Sets are Finitely Generated

We show that, while a smooth periodic set $\mathbf{P}$ may not be finitely generated, the set $\mathrm{Fill}(\mathbf{P})$ always is. We start with the following lemma.

▶ **Lemma 3.4.** *Let* $\mathbf{P}$ *be a periodic set. Then* $\mathrm{int}(\overline{\mathbb{Q}_{\geq 0}\mathbf{P}}) \subseteq \mathbb{Q}_{\geq 0}\mathbf{P} \subseteq \mathrm{dir}(\mathbf{P}) \subseteq \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$*.*
In particular, all these sets have the same closure.

Proof. Let $\mathbf{x} \in \mathrm{int}(\overline{\mathbf{C}})$, where $\mathbf{C} := \mathbb{Q}_{\geq 0}\mathbf{P}$. Then there exists $\varepsilon > 0$ such that the open ball $B(\mathbf{x}, \varepsilon)$ of radius ε around $\mathbf{x}$ is contained in $\overline{\mathbf{C}}$ by definition of interior. Hence for every $\mathbf{y} \in B(\mathbf{x}, \frac{\varepsilon}{2})$, there exists $f(\mathbf{y}) \in B(\mathbf{y}, \frac{\varepsilon}{4}) \cap \mathbf{C}$ by definition of closure. We have surrounded $\mathbf{x}$ by points $f(\mathbf{y}) \in \mathbf{C}$, hence by convexity of $\mathbf{C}$ we have $\mathbf{x} \in \mathbf{C}$.

Let $\mathbf{d} \in \mathbb{Q}_{\geq 0}\mathbf{P}$. Then there exists $m \in \mathbb{N}$ such that $m\mathbf{d} \in \mathbf{P}$, in particular $\mathbb{N} \cdot m\mathbf{d} \subseteq \mathbf{P}$.

Let $\mathbf{d} \in \operatorname{dir}(\mathbf{P})$. Then by replacing $\mathbf{d}$ by a multiple $m\mathbf{d}$, there exists $\mathbf{x}$ such that $\mathbf{x} + \mathbb{N} \cdot \mathbf{d} \subseteq \mathbf{P}$. We define the sequence $(\mathbf{x}_m)_{m \in \mathbb{N}}$ via $\mathbf{x}_m := \frac{1}{m}(\mathbf{x} + m \cdot \mathbf{d}) \in \mathbb{Q}_{\geq 0}\mathbf{P}$, and observe that its limit is $\mathbf{d}$, i.e. $\mathbf{d} \in \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$. ◀

▶ **Example 3.5.** The set on the left of Figure 2 satisfies $\operatorname{int}(\overline{\mathbb{Q}_{\geq 0}\mathbf{P}}) = \mathbb{Q}_{\geq 0}\mathbf{P} \subsetneq \operatorname{dir}(\mathbf{P}) = \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$. Indeed, $\operatorname{int}(\overline{\mathbb{Q}_{\geq 0}\mathbf{P}})$ contains every direction except north and east, but they both belong to $\operatorname{dir}(\mathbf{P})$. The middle set satisfies $\operatorname{int}(\overline{\mathbb{Q}_{\geq 0}\mathbf{P}}) \subsetneq \mathbb{Q}_{\geq 0}\mathbf{P} = \operatorname{dir}(\mathbf{P}) \subsetneq \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$, since $\operatorname{int}(\overline{\mathbb{Q}_{\geq 0}\mathbf{P}})$ contains neither north nor east, $\operatorname{dir}(\mathbf{P})$ contains east, and $\overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$ contains both.

We are now ready to reprove the result:

▶ **Proposition 3.6** ([11, Lemma 5.1]). *Let* $\mathbf{P}$ *be smooth. Then* $\operatorname{Fill}(\mathbf{P})$ *is full and hence f.g.*

Proof. Since $\mathbf{P}$ is smooth, $\operatorname{dir}(\mathbf{P})$ is definable by definition. By Lemma 3.4 we have $\overline{\mathbb{Q}_{\geq 0}\mathbf{P}} = \operatorname{dir}(\mathbf{P})$. So $\overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$ is the closure of a definable cone, and hence finitely generated by Lemma 2.1. Hence $\mathbf{P} = \operatorname{Fill}(\mathbf{P})$ is the intersection of a f.g. cone and a lattice, and hence full. ◀

3.2 Underapproximating Periodic Sets

In Section 3.1 we have seen that smooth periodic sets can be overapproximated by full linear sets in a natural way. Let us combine this with an underapproximation, mainly to provide a formal basis for the boundary function intuition above.

▶ **Proposition 3.7** ([13, Lemma F.1]). *Let* $\mathbf{P}$ *be a periodic set. Let* $\mathbf{F} \subseteq \mathbb{Q}^n$ *finite.*
$\mathbf{F} \subseteq (\mathbf{P} - \mathbf{P}) \cap \operatorname{dir}(\mathbf{P})$ *if and only if there exists* $\mathbf{x}$ *such that* $\mathbf{x} + \mathbf{F}^* \subseteq \mathbf{P}$.

Now consider any finitely generated cone $\mathbf{C} \subseteq \operatorname{dir}(\mathbf{P})$. Then $\mathbf{C} \cap (\mathbf{P} - \mathbf{P})$ is full and hence finitely generated by some set $\mathbf{F}$. By applying Proposition 3.7, we obtain a vector $\mathbf{x}_{\mathbf{C}} \in \mathbf{P}$ such that $\mathbf{x}_{\mathbf{C}} + (\mathbf{C} \cap (\mathbf{P} - \mathbf{P})) \subseteq \mathbf{P}$. This should be viewed as follows: Interpret the lattice $\mathbf{P} - \mathbf{P}$ as the set of "candidates" for being in $\mathbf{P}$. Namely, since $\mathbf{x}_{\mathbf{C}} \in \mathbf{P}$, a vector $\mathbf{x}_{\mathbf{C}} + \mathbf{v}$ can only be in $\mathbf{P}$ if $\mathbf{v} \in \mathbf{P} - \mathbf{P}$. Then $\mathbf{x}_{\mathbf{C}} + (\mathbf{C} \cap (\mathbf{P} - \mathbf{P})) \subseteq \mathbf{P}$ shows that every candidate in the given shifted cone (base point non-zero, so strictly speaking not a cone according to our definition) is actually in $\mathbf{P}$. Repeating this process for larger and larger cones $\mathbf{C}$, we obtain an underapproximation of $\mathbf{P}$ of the form $\bigcup_{f.g.\ \mathbf{C}} (\mathbf{x}_{\mathbf{C}} + \mathbf{C}) \cap (\mathbf{P} - \mathbf{P})$. The union of wider and wider shifted cones intuitively has a convex function as upper and a concave function as lower bound, as shown in the right of Figure 2.

Observe that this lower bound did not use smoothness, in general this might hence be a strict underapproximation, as shown in the right of Figure 2.

3.3 Intersection of Smooth Sets

We would like smooth sets to be closed under intersection. Further, we would like that the fill of an intersection of smooth sets is the intersection of the fills. However, this does not hold in general. The following is a counterexample.

▶ **Example 3.8.** Define $\mathbf{P} := \{\mathbf{0}\} \cup \mathbb{N}_{>0}^2$, see left of Figure 2, and $\mathbf{P}' = \{(0,1)\}^*$, the y-axis. We have $\{\mathbf{0}\} = \operatorname{dir}(\mathbf{P} \cap \mathbf{P}') \subsetneq \operatorname{dir}(\mathbf{P}) \cap \operatorname{dir}(\mathbf{P}')$. Also, $\{\mathbf{0}\} = \operatorname{Fill}(\mathbf{P} \cap \mathbf{P}') \subsetneq \operatorname{Fill}(\mathbf{P}) \cap \operatorname{Fill}(\mathbf{P}') = \mathbf{P}'$.

Fortunately, we can prove (see the Full version): Smooth sets $\mathbf{P}, \mathbf{P}'$ such that $\operatorname{Fill}(\mathbf{P})$, $\operatorname{Fill}(\mathbf{P}')$, and $\operatorname{Fill}(\mathbf{P}) \cap \operatorname{Fill}(\mathbf{P}')$ have the same dimension behave well under intersection.

▶ **Proposition 3.9.** *Let* $\mathbf{P}, \mathbf{P}'$ *be smooth periodic sets such that* $\dim(\mathrm{Fill}(\mathbf{P}) \cap \mathrm{Fill}(\mathbf{P}')) = \dim(\mathrm{Fill}(\mathbf{P})) = \dim(\mathrm{Fill}(\mathbf{P}'))$. *Then*

1. $\dim(\mathbf{P} \cap \mathbf{P}') = \dim(\mathbf{P}) = \dim(\mathbf{P}')$.
2. $\mathrm{dir}(\mathbf{P} \cap \mathbf{P}') = \mathrm{dir}(\mathbf{P}) \cap \mathrm{dir}(\mathbf{P}')$.
3. $\mathrm{Fill}(\mathbf{P} \cap \mathbf{P}') = \mathrm{Fill}(\mathbf{P}) \cap \mathrm{Fill}(\mathbf{P}')$.
4. $\mathbf{P} \cap \mathbf{P}'$ *is smooth.*

4 Petri sets and Hybridizations

We introduce the remaining classes of sets used in our main result: Petri sets and sets admitting a hybridization. Petri sets were introduced in [11–13]. Hybridizations are a novel notion, and play a fundamental role in our main result.

4.1 Petri sets

Leroux introduced almost semilinear sets and developed their theory in [12, 13]. Intuitively, they generalize semilinear sets by replacing linear sets with smooth periodic sets.

▶ **Definition 4.1** ([12, 13]). *A set* $\mathbf{X}$ *is* almost linear *if* $\mathbf{X} = \mathbf{b} + \mathbf{P}$, *where* $\mathbf{b} \in \mathbb{N}^n$ *and* $\mathbf{P}$ *is a smooth periodic set, and* almost semilinear *if it is a finite union of almost linear sets.*

It was shown in [12, 13] that VAS reachability sets are almost semilinear. However, it is easy to find almost semilinear sets that are not reachability sets of any VAS. Intuitively, the definition of a smooth periodic set only restricts the "asymptotic behavior" of the set, which can be "simple" even if the set itself is very "complex".

▶ **Example 4.2.** Let $\mathbf{Y} \subseteq \mathbb{N}_{>0}$ be any set. Then $\mathbf{P} := \{(0,0)\} \cup (\{1\} \times \mathbf{Y}) \cup \mathbb{N}^2_{>1}$ is a smooth periodic set; indeed, $\mathbf{P}$ contains a line in every direction, and is thus well-directed and asymptotically definable. So $\mathbf{P}$ is almost semilinear.

A way to eliminate at least some of these sets is to require that every intersection of the set with a semilinear set is still almost semilinear, a property enjoyed by all VAS reachability sets. For instance, assume that in Example 4.2 the set $\mathbf{Y}$ is not almost semilinear. Since the intersection of $\mathbf{P}$ and the linear set $(1,0) + (0,1) \cdot \mathbb{N}$ is equal to $\mathbf{Y}$, we can eliminate $\mathbf{P}$. This idea leads to the notion of a Petri set.

▶ **Definition 4.3** ([12, 13]). *A set* $\mathbf{X}$ *is called a* Petri set *if every intersection* $\mathbf{X} \cap \mathbf{S}$ *with a semilinear set* $\mathbf{S}$ *is almost semilinear.*

All smooth periodic sets shown so far are also Petri sets. To see that the positive examples are indeed Petri sets we can use the following strong theorem from [13].

▶ **Theorem 4.4** ([13, Theorem IX.1]). *Reachability sets of VAS are Petri sets.*

Many sets of the form $\{(x, y) \mid y \leq f(x)\}$ for convex f, or $\{(x, y) \mid y \geq f(x)\}$ for concave f, and boolean combinations thereof, are VAS reachability sets, and hence Petri sets.

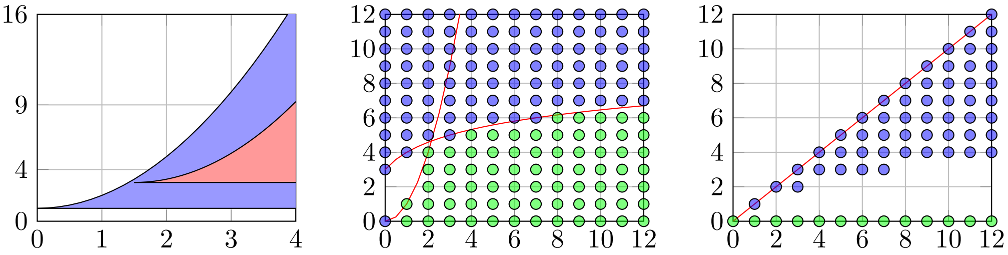


Figure 3 *Left*: An almost linear set $\mathbf{X} = \mathbf{b} + \mathbf{P}$ with $\mathbf{b} = (0, 1)$ and $\mathbf{P} = \{(x, y) \mid y \le x^2\}$ (in blue). The property $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X}$ implies that the "translation" of $\mathbf{X}$ to *any* point in the set (shown in brown for a particular point) is included in the set.
Middle: The two smooth periodic sets $\mathbf{P}_1 := \{(x, y) \mid y \ge \log_2(x + 1) + 3\} \cup \{(0, 0)\}$ in blue and $\mathbf{P}_2 := \{(x, y) \mid y \le x^2\}$ in green. Their union is almost hybridlinear, but not almost linear.
Right: The smooth periodic sets $\mathbf{P}_1 := \{(x, y) \mid x \ge y \ge \log_2(x+1)\}$ and $\mathbf{P}_2 := \{(1, 0)\}^*$. The union $\mathbf{X}$ does not have a hybridization, since $\mathbf{P} = \{(0, 0)\}$ is the only possibility to fulfill $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X}$.

4.2 Hybridizations

Given a Petri set $\mathbf{X} \subseteq \mathbb{N}^n$, it would be very useful to be able to partition $\mathbb{N}^n$ into finitely many semilinear regions $\mathbf{S}_1, \dots, \mathbf{S}_k$ such that the sets $\mathbf{S_i} \cap \mathbf{X}$ have a simpler structure. In particular, we would like $\mathbf{S_i} \ \cap \mathbf{X}$ to be almost linear. Unfortunately, for some Petri sets no such partition exists (an example can be found in the full version of the paper). We replace almost linearity by a slightly weaker notion for which the partition always exists: having a hybridization (Definition 4.5).

A set is almost linear if there exists a vector $\mathbf{b}$ and a smooth periodic set $\mathbf{P}$ such that $\mathbf{X} = \mathbf{b} + \mathbf{P}$. The following definition is equivalent: There exists a vector $\mathbf{b}$ and a smooth periodic set $\mathbf{P}$ such that $\mathbf{b} \in \mathbf{X}$ and $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X} \subseteq \mathbf{b} + \mathbf{P}$.

We weaken this condition by requiring only the existence of a vector $\mathbf{b}$ and a smooth periodic set $\mathbf{P}$ such that $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X} \subseteq \mathbf{b} + \mathrm{Fill}(\mathbf{P})$.

That is, we drop the condition $\mathbf{b} \in \mathbf{X}$, and replace $\mathbf{P}$ on the right by the possibly larger set $\mathrm{Fill}(\mathbf{P})$. (For example, the periodic sets on the left of Figure 3 as well as in the middle satisfy $\mathrm{Fill}(\mathbf{P}) = \mathbb{N}^2$). We then call the set $\mathbf{b} + \mathrm{Fill}(\mathbf{P})$ a hybridization of $\mathbf{X}$. The formal definition is as follows, where for technical reasons we also introduce weak hybridizations.

▶ **Definition 4.5.** *Let $\mathbf{X} \subseteq \mathbb{N}^n$ be non-empty. A set $\mathbf{H}$ is a* weak hybridization *of $\mathbf{X}$ if there exists a finite set $\mathbf{B} \subseteq \mathbb{N}^n$ and a smooth periodic set $\mathbf{P}$ such that $\mathbf{H} = \mathbf{B} + \mathrm{Fill}(\mathbf{P})$ and $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X} \subseteq \mathbf{H}$. If $\mathbf{B} = \{\mathbf{b}\}$, then $\mathbf{H}$ is a* hybridization *of $\mathbf{X}$.*

▶ Remark 4.6. There are full linear weak hybridizations which are not hybridizations. For example $\mathbf{X} = 1 + 3\mathbb{N} \cup 2 + 3\mathbb{N}$ has weak hybridization $\mathbf{H} = \{0, 1, 2\} + 3\mathbb{N} = \mathbb{N}$. However, since $\mathbf{X}$ does not contain any points congruent to 0 modulo 3, any periodic set $\mathbf{P}$ fulfilling $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X}$ has to fulfill $\mathbf{P} \subseteq 3\mathbb{N}$. Hence $\mathbf{B}$ cannot be chosen as a singleton.

It follows from this definition that almost linear sets have hybridizations. The reason for the name (weak) hybridization is that the set $\mathbf{H}$ is always hybridlinear, a notion introduced in [4] by Ginsburg and Spanier and later studied in [1] by Chistikov and Haase. We recall the definition for future reference.

▶ **Definition 4.7.** *A set $\mathbf{H} \subseteq \mathbb{N}^n$ is* hybridlinear *if $\mathbf{H} = \mathbf{B} + \mathbf{P}$ for some finite set $\mathbf{B}$ and some finitely generated periodic set $\mathbf{P} \subseteq \mathbb{N}^n$.*

We end this section with a characterization of the sets that admit weak hybridizations.

▶ **Definition 4.8.** *A non-empty set* $\mathbf{X} \subseteq \mathbb{N}^n$ *is* almost hybridlinear *if there exist* $\mathbf{b}_1, \ldots, \mathbf{b}_r \in \mathbb{N}^n$ *and smooth* $\mathbf{P}_1, \ldots, \mathbf{P}_r$ *with* $\mathbf{X} = \bigcup_{i=1}^r \mathbf{b}_i + \mathbf{P}_i$*, such that* $\text{Fill}(\mathbf{P}_i) = \text{Fill}(\mathbf{P}_j)$ *for all* i, j*.*

▶ **Theorem 4.9.** *A non-empty Petri set* $\mathbf{X} \subseteq \mathbb{N}^n$ *is almost hybridlinear if and only if it has a weak hybridization.*

This theorem helps to find examples of non-trivial hybridizations (i.e. not of type $\mathbf{P}$ has hybridization $\text{Fill}(\mathbf{P})$). For example $[(0,1) + \mathbf{P}_1] \cup [(0,6) + \mathbf{P}_2]$ for $\mathbf{P}_1 = \{(x,y) \in \mathbb{N}^2 \mid y \leq x^2\}$ and $\mathbf{P}_2 = \{(x,y) \in \mathbb{N}^2 \mid y \geq \log_2(x+1)\}$ has weak hybridization $\mathbb{N}^2$, since $\text{Fill}(\mathbf{P}_1) = \text{Fill}(\mathbf{P}_2) = \mathbb{N}^2$. This is very similar to the middle of Figure 3. On the other hand, in the right of Figure 3 the smooth periodic sets barely intersect, and then the union is usually not almost hybridlinear.

5 Proof of Theorem 1.1

In this section we prove Theorem 1.1. The algorithm and its proof will refine the partition in three steps, respectively described in Section 5.1, Section 5.2 and Section 5.3: During the first two steps the sets $\mathbf{X} \cap \mathbf{S}_i$ are not required to be irreducible, and in addition after the first step, the $\mathbf{S}_i$ are allowed to be hybridlinear instead of full linear.

5.1 Existence of a Hybridlinear Partition

We collect five important properties of (weak) hybridizations in Proposition 5.2. Then, we use these properties to formulate a procedure for producing a partition $\mathbf{S} = \mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ of sets, not necessarily full linear, satisfying the properties of Theorem 1.1 except for irreducibility. The procedure is described in Figure 4. It is effective for VAS reachability sets, but not in general.

We start by reminding that the class of hybridlinear sets is closed under intersection.

▶ **Lemma 5.1** ([10, Lemma 7.8])**.** *Let* $\mathbf{b}_1 + \mathbf{Q}_1$ *and* $\mathbf{b}_2 + \mathbf{Q}_2$ *be linear sets. Then* $(\mathbf{b}_1 + \mathbf{Q}_1) \cap (\mathbf{b}_2 + \mathbf{Q}_2) = \mathbf{B} + (\mathbf{Q}_1 \cap \mathbf{Q}_2)$ *for some finite* $\mathbf{B}$*.*

▶ **Proposition 5.2.** *The following statements hold:*

1) *If* $\mathbf{H}$ *is a weak hybridization of* $\mathbf{X}$*, then* $\dim(\mathbf{X}) = \dim(\mathbf{H})$*.*

2) *If* $\mathbf{H}$ *is a weak hybridization of* $\mathbf{X}$ *and* $\mathbf{L} = \mathbf{b} + \mathbf{Q}$ *full linear s.t.* $\dim(\mathbf{H} \cap \mathbf{L}) = \dim(\mathbf{H}) = \dim(\mathbf{L})$*, then* $\mathbf{H} \cap \mathbf{L}$ *is a weak hybridization for* $\mathbf{X} \cap \mathbf{L}$*, or* $\mathbf{X} \cap \mathbf{L}$ *is empty.*

3) *If* $\mathbf{H}$ *is a (weak) hybridization for both* $\mathbf{X}_1$ *and* $\mathbf{X}_2$*, then* $\mathbf{H}$ *is a (weak) hybridization for* $\mathbf{X}_1 \cup \mathbf{X}_2$*.*

4) *For every Petri set* $\mathbf{X}$ *and semilinear* $\mathbf{S}$ *there is a partition* $\mathbf{X} \cap \mathbf{S} = \mathbf{X}_1 \cup \cdots \cup \mathbf{X}_r$ *of* $\mathbf{X} \cap \mathbf{S}$ *such that every* $\mathbf{X}_i$ *has a (true) hybridization* $\mathbf{L}_i$*.*

5) *If* $\mathbf{X}$ *is the reachability set of a VAS, then the set* $\{\mathbf{L}_1, \ldots, \mathbf{L}_r\}$ *of hybridizations of part 4) is computable.*

Proof. For proofs 1) and 2), write $\mathbf{H} := \mathbf{B} + \text{Fill}(\mathbf{P})$, where $\mathbf{P}$ is smooth and $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X}$.

1): This follows from the properties of dimension in Lemmas 2.4 and 2.5. In particular, $\dim(\mathbf{P}) = \dim(\mathbf{V})$, where $\mathbf{V}$ is the vector space generated by $\mathbf{P}$, also implies $\dim(\mathbf{P}) = \dim(\text{Fill}(\mathbf{P}))$. Hence $\mathbf{X} \subseteq \mathbf{H}$ implies $\dim(\mathbf{X}) \leq \dim(\mathbf{P})$. Since $\mathbf{X}$ is non-empty, $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X}$ implies $\dim(\mathbf{X}) \geq \dim(\mathbf{P})$.

Partition$(\mathbf{X}, \mathbf{S})$. Input: Petri set $\mathbf{X}$ and semilinear set $\mathbf{S}$:

1) If $\mathbf{S}$ is empty, return $\mathbf{S}$. If $\mathbf{S}$ is not full, compute a partition $\mathbf{S}_1, \dots, \mathbf{S}_r$ of $\mathbf{S}$ into full linear sets, return $\bigcup_{i=1}^{r}$ Partition$(\mathbf{X}, \mathbf{S}_i)$ and stop.

Otherwise, compute the set $\mathcal{L} = \{\mathbf{L}_1, \dots, \mathbf{L}_r\}$ of hybridizations of the partition $\mathbf{X}_1 \cup \cdots \cup \mathbf{X}_r$ of $\mathbf{X} \cap \mathbf{S}$ given by Proposition 5.2(4), and move to step 2).

Remark: This step is not effective for arbitrary Petri sets, but it is effective for VAS reachability sets by Proposition 5.2(5).

If $r = 0$, i.e., if $\mathbf{X} \cap \mathbf{S}$ is empty, then return $\mathbf{S}$ and stop. Otherwise, move to step 2).

2) For every $\mathbf{L}_i \in \mathcal{L}$ compute a decomposition $\mathcal{K}_i$ of $\mathbf{L}_i^C \cap \mathbf{S}$ into full linear sets, where $\mathbf{L}_i^C$ is the complement of $\mathbf{L}_i$, and move to step 3).

3) Let $\mathcal{M}$ be the set of tuples $(\mathbf{M}_1, \dots, \mathbf{M}_r) \in (\{\mathbf{L}_1\} \cup \mathcal{K}_1) \times \cdots \times (\{\mathbf{L}_r\} \cup \mathcal{K}_r)$.
For every $M \in \mathcal{M}$, let $\mathbf{S}_M := \mathbf{S} \cap \mathbf{M}_1 \cap \cdots \cap \mathbf{M}_r$.

Remark: $\{\mathbf{S}_M \mid M \in \mathcal{M}\}$ *is a partition of* $\mathbf{S}$.

For every $M \in \mathcal{M}$, define P_M as follows: If $\dim(\mathbf{S}_M) < \dim(\mathbf{S})$, then $P_M :=$ Partition$(\mathbf{X}, \mathbf{S}_M)$, otherwise $P_M := \{\mathbf{S}_M\}$. Output $\bigcup_{M \in \mathcal{M}} P_M$.

Figure 4 The procedure Partition$(\mathbf{X}, \mathbf{S})$.

2): By Lemma 5.1, $\mathbf{H} \cap \mathbf{L} = \mathbf{F} + (\text{Fill}(\mathbf{P}) \cap \mathbf{Q})$ for some finite set $\mathbf{F}$. By Proposition 3.9, we have that $\mathbf{P} \cap \mathbf{Q}$ is smooth and $\text{Fill}(\mathbf{P} \cap \mathbf{Q}) = \text{Fill}(\mathbf{P}) \cap \text{Fill}(\mathbf{Q}) = \text{Fill}(\mathbf{P}) \cap \mathbf{Q}$. We have $\mathbf{X} \cap \mathbf{L} \subseteq \mathbf{H} \cap \mathbf{L}$. We also have $(\mathbf{X} \cap \mathbf{L}) + (\mathbf{P} \cap \mathbf{Q}) \subseteq \mathbf{X} + \mathbf{P} \subseteq \mathbf{X}$ and $(\mathbf{X} \cap \mathbf{L}) + (\mathbf{P} \cap \mathbf{Q}) \subseteq \mathbf{L} + \mathbf{Q} \subseteq \mathbf{L}$, hence $\mathbf{H} \cap \mathbf{L}$ is a weak hybridization of $\mathbf{X} \cap \mathbf{L}$.

3): Write $\mathbf{B}_1 + \text{Fill}(\mathbf{P}_1) = \mathbf{H} = \mathbf{B}_2 + \text{Fill}(\mathbf{P}_2)$, where $\mathbf{P}_1$ for $\mathbf{X}_1$ and $\mathbf{P}_2$ for $\mathbf{X}_2$ are as in the definition of weak hybridization. By Lemma 5.1, we have $\mathbf{H} = \mathbf{H} \cap \mathbf{H} = \mathbf{F} + [\text{Fill}(\mathbf{P}_1) \cap \text{Fill}(\mathbf{P}_2)]$ for some finite set $\mathbf{F}$. Define $\mathbf{P} := \mathbf{P}_1 \cap \mathbf{P}_2$ and $\mathbf{X} := \mathbf{X}_1 \cup \mathbf{X}_2$. By Proposition 3.9, $\mathbf{P}$ is smooth and $\text{Fill}(\mathbf{P}) = \text{Fill}(\mathbf{P}_1) \cap \text{Fill}(\mathbf{P}_2)$. We also have $\mathbf{X} + \mathbf{P} \subseteq \mathbf{X}$.

4): Since $\mathbf{X}$ is a Petri set, $\mathbf{X} \cap \mathbf{S}$ is almost semilinear, and can hence be written as $\mathbf{X} = \bigcup_{i=1}^{r} \mathbf{b}_i + \mathbf{P}_i$ for smooth periodic sets $\mathbf{P}_i \subseteq \mathbb{N}^n$ and points $\mathbf{b}_i \in \mathbb{N}^n$. Every $\mathbf{X}_i := \mathbf{b}_i + \mathbf{P}_i$ is by definition almost hybridlinear with hybridization $\mathbf{b}_i + \text{Fill}(\mathbf{P}_i)$, which is a full linear set.

5): 4) can be computed using the Kosaraju-Lambert-Mayr-Sacerdote-Tenney (KLMST) decomposition [8–10, 16]. The KLMST decomposition constructs a finite set of VASS-like objects, called perfect marked graph transition sequences or perfect MGTSs, such that the set of reachable configurations of the VAS is the union of the sets of reachable configurations of the perfect MGTSs. Further, for every perfect MGTS one can effectively construct a set of linear equations satisfying the following property: the set of solutions of the equation system is a hybridization of the set of reachable configurations of the perfect MGTS. The set of solutions of a system of linear equations is always hybridlinear. Moreover, for the systems derived from MGTSs one can show that the set has a full linear hybridization (e.g. [10, Lemma 5.1]). This gives us the desired hybridizations $\mathbf{L}_1, \dots, \mathbf{L}_r$. [2] ◀

[2] While Hauschildt already used the KLMST decomposition in [6] in 1990, it took until 2019 [15, 16] to fully understand the theoretical aspects behind the algorithm and its complexity of Ackermann.

▶ **Proposition 5.3.** *Let* $\mathbf{X}$ *be a Petri set and let* $\mathbf{S}$ *be a semilinear set. Partition*$(\mathbf{X}, \mathbf{S})$ *produces a partition* $\mathbf{S} = \mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ *into pairwise disjoint hybridlinear sets (not necessarily full linear) such that for every* i *the set* $\mathbf{X} \cap \mathbf{S}_i$ *is either empty or has weak hybridization* $\mathbf{S}_i$*. Further, if* $\mathbf{X}$ *is the reachability set of a VAS, then the partition is computable.*

Proof. The procedure is depicted in Figure 4, in addition we give an intuitive description of it: In Step 1) we first partition $\mathbf{S}$ into full linear sets and consider them separately. So assume that $\mathbf{S}$ is a full linear set. The procedure uses Proposition 5.2(5) to compute a set of full linear hybridizations $\mathbf{L}_1, \dots, \mathbf{L}_r$ of a partition $\mathbf{X}_1 \cup \cdots \cup \mathbf{X}_r$ of $\mathbf{X} \cap \mathbf{S}$. Step 2) considers all possible sets obtained by picking for each $i \in \{1, \dots, r\}$ either the set $\mathbf{L}_i$ or a linear set of its complement (its complement is semilinear, and so a finite union of linear sets), and intersecting all of them. The procedure adds all the sets having full dimension to the output partition, and does a recursive call on the others.

Every step can be performed: The set $\mathcal{L}$ of Step 1 exists by Proposition 5.2(4). To check the dimension of a semilinear set $\mathbf{S} = \bigcup_{j=1}^{r} \mathbf{b}_j + \mathbf{F}_j^*$, which is needed in step 3), we use Lemma 2.5 to obtain that for $\mathbf{F}_j^*$ this is simply the rank of the generator matrix, and by Lemma 2.4 we have $\dim(\mathbf{S}) = \max_j \dim(\mathbf{F}_j^*)$.

Termination: Partition$(\mathbf{X}, \mathbf{S})$ only performs a recursive call if $\mathbf{S}$ is not a full linear set or on semilinear sets $\mathbf{S}'$ with $\dim(\mathbf{S}') < \dim(\mathbf{S})$, hence recursion depth is at most $2\dim(\mathbf{S}) + 1$ and termination immediate.

Correctness: The proof obligation for correctness is that for every $M = (\mathbf{M}_1, \dots, \mathbf{M}_r) \in \mathcal{M}$, where S_M fulfills $\dim(\mathbf{S}_M) = \dim(\mathbf{S})$, $\mathbf{X} \cap \mathbf{S}_M$ is either empty or has $\mathbf{S}_M$ as weak hybridization. Therefore fix such M.

▷ Claim 5.4. $\dim(\mathbf{M}_j) = \dim(\mathbf{S})$ for all j.

Proof of Claim. $\geq \dim(\mathbf{S})$ follows since all these sets contain $\mathbf{S}_M$, which fulfills $\dim(\mathbf{S}_M) = \dim(\mathbf{S})$. For the other direction, to prove "$\leq$" for j where we choose $\mathbf{L}_j$ we have $\dim(\mathbf{S}) \geq \dim(\mathbf{X} \cap \mathbf{S}) = \max_j \dim(\mathbf{L}_j)$ by Proposition 5.2. For other j we use $\mathbf{L}_j^C \cap \mathbf{S} \subseteq \mathbf{S}$. ◁

The claim allows us to use Proposition 5.2(2). Let $\mathbf{X}_j$ be such that $\mathbf{X} \cap \mathbf{S} = \bigcup_{j=1}^{r} \mathbf{X}_j$ and $\mathbf{X}_j$ has hybridization $\mathbf{L}_j$. By applying Proposition 5.2(2) enough times, for every j with $\mathbf{M}_j = \mathbf{L}_j$, we obtain that $\mathbf{X}_j \cap \mathbf{S}_M$ has weak hybridization $\mathbf{S}_M$. This does not depend on j because intersecting with $\mathbf{L}_j$ twice does not change the set. For all other j we have $\mathbf{X}_j \cap \mathbf{S}_M = \emptyset$, since we intersect with the complement of an overapproximation. Hence $\mathbf{X} \cap \mathbf{S}_M = \bigcup_{j, \mathbf{M}_j = \mathbf{L}_j} (\mathbf{X}_j \cap \mathbf{S}_M)$ has weak hybridization $\mathbf{S}_M$ by Proposition 5.2(3), or is empty if we never chose $\mathbf{M}_j = \mathbf{L}_j$. ◀

5.2 Existence of a Full Linear Partition

We show that Proposition 5.3 can be strengthened to make the sets $\mathbf{S}_i$ not only hybridlinear, but even full linear, in a way that the sets $\mathbf{S}_i$ are actually (true) hybridizations.

▶ **Proposition 5.5.** *Let* $\mathbf{X}$ *be a Petri set. For every semilinear set* $\mathbf{S}$ *there exists a partition* $\mathbf{S} = \mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ *of* $\mathbf{S}$ *into pairwise disjoint full linear sets such that for every* i *the set* $X \cap \mathbf{S}_i$ *is either empty or has hybridization* $\mathbf{S}_i$*. Further, if* $\mathbf{X}$ *is the reachability set of a VAS, then the partition is computable.*

Proof. The main algorithm uses a subroutine with the same inputs and outputs as itself, but with the promise that $\mathbf{X} \cap \mathbf{S}$ has weak hybridization $\mathbf{S}$. We first describe the main algorithm, and then the subroutine.

Main algorithm: First apply Proposition 5.3 to obtain a partition $\mathbf{S} = \mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ into hybridlinear sets otherwise satisfying the conditions. Output $\bigcup_{i=1}^k \text{Subroutine}(\mathbf{X}, \mathbf{S}_i)$.

Subroutine: If $\mathbf{S}$ is already full linear, return $\mathbf{S}$. Otherwise write $\mathbf{S} = \{\mathbf{c}_1, \dots, \mathbf{c}_r\} + \text{Fill}(\mathbf{P})$. Let $j \sim k \iff \mathbf{c}_j - \mathbf{c}_k \in \text{Fill}(\mathbf{P}) - \text{Fill}(\mathbf{P}) = \mathbf{P} - \mathbf{P}$. Compute a system R of representatives for $\sim$. For every $i \in R$, define $\mathbf{S}_i := \mathbf{c}_i + \text{Fill}(\mathbf{P})$. Define $\mathbf{S}' := \mathbf{S} \setminus \bigcup_{i \in R} \mathbf{S}_i$ and output $\{\mathbf{S}_i \mid i \in R\} \cup \text{MainAlgorithm}(\mathbf{X}, \mathbf{S}')$.

Termination: We prove that recursion depth $\leq 2\dim(\mathbf{S}) + 1$ by proving $\dim(\mathbf{S}') < \dim(\mathbf{S})$ in the subroutine. For every equivalence class C of $\sim$, there exists $\mathbf{c} \in \mathbb{Z}^n$ such that $\mathbf{c_j} - \mathbf{c} \in \mathbf{P}$ for all $j \in C$. To see this, fix some $i \in C$, and write $\mathbf{c}_j - \mathbf{c}_i = \mathbf{p}_j - \mathbf{p}'_j \in \mathbf{P} - \mathbf{P}$. Choose $\mathbf{c} := \mathbf{c}_i - \sum_{j \in C} \mathbf{p}'_j$.

Then $\bigcup_{j \in C} \mathbf{c}_j + \text{Fill}(\mathbf{P}) \subseteq \mathbf{c} + \text{Fill}(\mathbf{P})$, and hence using Lemma 2.10 we obtain
$\dim(\bigcup_{j \in C} \mathbf{c}_j + \text{Fill}(\mathbf{P}) \setminus \mathbf{S}_i) \leq \dim(\mathbf{c} + \text{Fill}(\mathbf{P}) \setminus \mathbf{c}_i + \text{Fill}(\mathbf{P})) < \dim(\text{Fill}(\mathbf{P}))$.

Correctness: The main algorithm is clearly correct if the subroutine is. In the subroutine, we have $\mathbf{S}_i \cap \mathbf{S}_j = \emptyset$ since $i \not\sim j$ for $i, j \in R$. All $\mathbf{S}_i$ are full linear by definition. Furthermore, $\mathbf{X} \cap \mathbf{S}_i$ has weak hybridization $\mathbf{H} \cap \mathbf{S}_i = \mathbf{S}_i$ by Proposition 5.2(2). To obtain that the hybridization is not weak, observe that Proposition 5.2(2) specifically shows that the intersection of the representations, which is the full linear representation of $\mathbf{S}_i$, is a weak hybridization. ◀

5.3 Reducibility of almost hybridlinear Sets

The final ingredient of our main result is reducibility. We name it after its counterpart in Hauschildt's PhD thesis [6].

▶ **Definition 5.6.** *A set* $\mathbf{X}$ *with hybridization* $\mathbf{c} + \text{Fill}(\mathbf{P})$ *is* reducible *if there exists* $\mathbf{x}$ *such that* $\mathbf{x} + \text{Fill}(\mathbf{P}) \subseteq \mathbf{X}$.

In other words, $\mathbf{X}$ is reducible if every large enough point of its hybridization is already in $\mathbf{X}$. Observe that this does not follow from hybridization, as $\text{Fill}(\mathbf{P})$ is larger than $\mathbf{P}$. Our usual examples of sets with hybridization are smooth periodic sets, these also illustrate reducibility: The set in the left of Figure 2 is reducible, while the middle is not. Another example of hybridization was in the middle of Figure 3, this set is also reducible. In fact, whenever $\mathbf{X} = \mathbf{b} + \mathbf{P}$, $\mathbf{X}$ is reducible if and only if $\text{dir}(\mathbf{P}) = \overline{\mathbb{Q}_{\geq 0}\mathbf{P}}$. Namely, use Proposition 3.7 with $\mathbf{F}$ the generators of $\text{Fill}(\mathbf{P})$. For other sets $\mathbf{X}$, write $\mathbf{X} = \bigcup_{i=1}^r \mathbf{b}_i + \mathbf{P}_i$ as almost hybridlinear set. Whether it is reducible again only depends on the cones $\text{dir}(\mathbf{P}_i)$, for a proof see the full version. Since matrices for the definable cones $\text{dir}(\mathbf{P}_i)$ can in the case of VAS be determined using KLMST-decomposition [6], we obtain the following.

▶ **Theorem 5.7** ([6, even without promise]). *The following problem is decidable.*
Input: Reachability set $\mathbf{R}$*, represented via the transitions of the VASS, full linear set* $\mathbf{S}$*.*
Promise: $\mathbf{R} \cap \mathbf{S}$ *has hybridization* $\mathbf{S}$*.*
Output: Is $\mathbf{R} \cap \mathbf{S}$ *reducible?*

We can now prove our main result.

▶ **Theorem 1.1.** *Let* $\mathbf{X}$ *be a Petri set. For every semilinear set* $\mathbf{S}$ *there exists a partition* $\mathbf{S} = \mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ *into pairwise disjoint full linear sets such that for all* $i \in \{1, \dots, k\}$ *either* $\mathbf{X} \cap \mathbf{S}_i = \emptyset$*,* $\mathbf{S}_i \subseteq \mathbf{X}$ *or* $\mathbf{X} \cap \mathbf{S}_i$ *is irreducible with hybridization* $\mathbf{S}_i$*. Further, if* $\mathbf{X}$ *is the reachability set of a VAS, then the partition is computable.*

Proof. Step 1: Use Proposition 5.5 to compute a partition $\mathbf{S} = \mathbf{S}_1 \cup \cdots \cup \mathbf{S}_k$ into full linear sets such that $\mathbf{X} \cap \mathbf{S}_i$ has hybridization $\mathbf{S}_i$ if it is non-empty. For every set $\mathbf{S}_i$ with $\mathbf{X} \cap \mathbf{S}_i \neq \emptyset$ do Step 2.

Step 2: Decide whether $\mathbf{X} \cap \mathbf{S}_i$ is reducible using Theorem 5.7. If irreducible, output $\mathbf{S}_i$. Otherwise, there exists $\mathbf{x}$ such that $\mathbf{x} + \mathbf{Q} \subseteq \mathbf{X} \cap \mathbf{S}_i$, where $\mathbf{S}_i = \mathbf{c} + \mathbf{Q}$. Find such an $\mathbf{x}$, add $\mathbf{x} + \mathbf{Q} \subseteq \mathbf{X}$ to the final partition and do a recursive call on $\mathbf{S}_i \setminus (\mathbf{x} + \mathbf{Q})$.

Termination: We claim that we only perform recursion on $\mathbf{S}'$ with $\dim(\mathbf{S}') < \dim(\mathbf{S})$. To see this, take $\mathbf{S}_i = \mathbf{c} + \mathbf{Q}$ such that $\mathbf{X} \cap \mathbf{S}_i$ is reducible. We have $\dim(\mathbf{S}_i \setminus \mathbf{x} + \mathbf{Q}) = \dim(\mathbf{c}+\mathbf{Q} \setminus \mathbf{x}+\mathbf{Q}) < \dim(\mathbf{Q})$ by Lemma 2.10, wherefore the recursion uses a lower dimensional set, and termination follows from bounded recursion depth.

Correctness: Follows from correctness of Proposition 5.5.

The partition is computable for VAS: We have to be able to find $\mathbf{x}$ with $\mathbf{x} + \mathbf{Q} \subseteq \mathbf{X}$ given the promise that such an $\mathbf{x}$ exists. This is possible since containment of semilinear sets in reachability sets is decidable by [13] using flatability. ◀

6 Corollaries of Theorem 1.1

6.1 VAS semilinearity is decidable

We reprove that the semilinearity problem for VAS is decidable. We start with a lemma, whose full proof is in the full version.

▶ **Lemma 6.1.** *Let $\mathbf{X}$ be a semilinear Petri set with hybridization $\mathbf{c} + \mathbf{Q}$. Then $\mathbf{X}$ is reducible.*

Proof idea. The hybridization describes all "limit directions", with the problematic ones being for example "north" in case of the parabola $\{(x, y) \mid y \leq x^2\}$, which is a limit but not actually a direction. If $\mathbf{X}$ is semilinear though, then the steepness can only increase finitely often, namely when changing to a different linear component, and all limit directions are actually also directions. Using this for generators of $\mathrm{Fill}(\mathbf{P})$ we find $\mathbf{x} + \mathrm{Fill}(\mathbf{P}) \subseteq \mathbf{X}$. ◀

▶ **Corollary 6.2** ([6]). *The following problem is decidable.*
Input: Reachability set $\mathbf{R}$ of VAS, semilinear $\mathbf{S}$.
Output: Is $\mathbf{R} \cap \mathbf{S}$ semilinear?

Proof. As also mentioned in the introduction, the algorithm computes the partition of Theorem 1.1 and checks whether the third case does not occur.

Correctness: If $\mathbf{R} \cap \mathbf{S}$ is semilinear, then in particular $\mathbf{R} \cap \mathbf{S}_i$ is semilinear for every part $\mathbf{S}_i$ of the partition. By Lemma 6.1, $\mathbf{R} \cap \mathbf{S}_i$ cannot be irreducible, and so either $\mathbf{R} \cap \mathbf{S}_i = \emptyset$ or $\mathbf{S}_i \subseteq \mathbf{R}$ for all i.

On the other hand, if only the cases $\mathbf{R} \cap \mathbf{S}_i = \emptyset$ and $\mathbf{S}_i \subseteq \mathbf{R}$ occur, then the $\mathbf{S}_i$ such that $\mathbf{S}_i \subseteq \mathbf{R}$ form a semilinear representation. ◀

6.2 On the Complement of a VAS Reachability Set

We show that if the complement of a VAS reachability set is infinite, then it contains an infinite linear set. The main part of the argument was already depicted in the middle of Figure 3: If $\mathbf{X}$ contains enough of the boundary, then it is reducible.

We hence need to formalize the notion of boundary and interior also for full linear sets. If $\mathbf{L} = \mathbf{b} + \mathbf{Q}$ is a full linear set, then $\mathrm{int}(\mathbf{L}) := \mathbf{b} + (\mathbf{Q} \cap \mathrm{int}(\mathbb{Q}_{\geq 0}\mathbf{Q}))$ is the interior of $\mathbf{L}$ and $\partial(\mathbf{L}) := \mathbf{b} + (\mathbf{Q} \cap \partial(\mathbb{Q}_{\geq 0}\mathbf{Q}))$ is the boundary of $\mathbf{L}$, both are inherited from the cone. These sets are both semilinear, as can be seen by using the definition expressible via $\varphi \in \mathrm{FO}(\mathbb{N}, +, \geq)$, i.e. Presburger Arithmetic. Remember that we consider definable cones, i.e. cones expressible in $\mathrm{FO}(\mathbb{Q}, +, \geq)$. In the full version, we prove the following proposition, formalizing the first part of the proof.

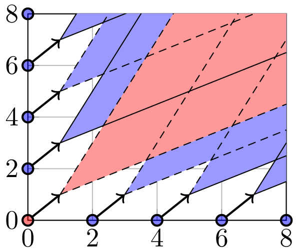


Figure 5 Let $\mathbf{C}$ be the cone generated by $(2,1)$ and $(1,2)$ and assume that $\mathbf{X} + [(1,1) + \mathbf{C}] \subseteq \mathbf{X}$ holds. Then $(0,0) \in \mathbf{X}$ implies that the whole red shifted cone is in $\mathbf{X}$. Importantly, we obtain a similar shifted cone for *every* point $\mathbf{x}' \in \mathbf{X}$. Hence if $\partial\mathbb{N}^2 \subseteq \mathbf{X}$, then almost all of $\mathbb{N}^2$ is contained in $\mathbf{X}$.

▶ **Proposition 6.3.** *Let* $\mathbf{X}$ *be a set with hybridization* $\mathbf{c} + \mathrm{Fill}(\mathbf{P})$*. Assume that* $|\partial(\mathbf{c} + \mathrm{Fill}(\mathbf{P})) \setminus \mathbf{X}| < \infty$*. Then* $\mathbf{X}$ *is reducible.*

The proof of Proposition 6.3 is illustrated in the above figure. The main difficulty is defining a “wide enough” cone $\mathbf{C}$, then Proposition 3.7 applied to $\mathbf{C} \cap (\mathbf{P} - \mathbf{P})$ does the rest.

▶ **Corollary 6.4.** *Let* $\mathbf{X}$ *be a Petri set. Let* $\mathbf{S}$ *be a semilinear set such that* $\mathbf{S} \setminus \mathbf{X}$ *is infinite. Then* $\mathbf{S} \setminus \mathbf{X}$ *contains an infinite linear set.*

Proof. Proof by induction on $\dim(\mathbf{S})$. If $\dim(\mathbf{S}) = 0$, the property holds vacuously. Else consider the partition of Theorem 1.1. Since $\mathbf{S} \setminus \mathbf{X}$ is infinite, some $\mathbf{S}_i \setminus \mathbf{X}$ is infinite. Fix such an i. Because of Theorem 1.1, $\mathbf{S}_i \subseteq \mathbf{X}$ or $\mathbf{X} \cap \mathbf{S}_i = \emptyset$ or $\mathbf{X} \cap \mathbf{S}_i$ is irreducible. In fact, only the third possibility is interesting. If $\mathbf{S}_i \subseteq \mathbf{X}$, then $\mathbf{S}_i \setminus \mathbf{X}$ can not be infinite. If $\mathbf{S}_i \cap \mathbf{X} = \emptyset$ then $\mathbf{S}_i = \mathbf{S}_i \setminus \mathbf{X}$, hence it contains a line. Let us consider the case when $\mathbf{S}_i \cap \mathbf{X}$ is irreducible. Assume for contradiction that $\mathbf{S}_i \setminus \mathbf{X}$ does not contain an infinite linear set. Then in particular $\partial(\mathbf{S}_i) \setminus \mathbf{X}$ does not. We have $\dim(\partial(\mathbf{S_i})) < \dim(\mathbf{S_i})$, since the boundary is contained in the finite union of the facets. Hence $|\partial(\mathbf{S}_i) \setminus \mathbf{X}| < \infty$ by induction. By Proposition 6.3, $\mathbf{X} \cap \mathbf{S}_i$ is reducible. Contradiction. ◀

In the full version, we even prove another corollary of the partition. The proof is based on the existence of a partition as in Theorem 1.1, which has the properties for two Petri sets $\mathbf{X}_1$ and $\mathbf{X}_2$ at once.

▶ **Corollary 6.5.** *Let* $\mathbf{X}_1$ *and* $\mathbf{X}_2$ *be Petri sets with* $\mathbf{X}_1 \cap \mathbf{X}_2 = \emptyset$*. Then there exists a semilinear set* $\mathbf{S}'$ *such that* $\mathbf{X}_1 \subseteq \mathbf{S}'$ *and* $\mathbf{X}_2 \cap \mathbf{S}' = \emptyset$*.*

▶ **Corollary 6.6.** *Let* $\mathcal{V}$ *be a VAS, and* $\mathbf{X}$ *a Petri set such that* $\mathrm{Reach}(\mathcal{V}) \cap \mathbf{X} = \emptyset$*. Then there exists a semilinear inductive invariant* $\mathbf{S}'$ *of* $\mathcal{V}$ *such that* $\mathrm{Reach}(\mathcal{V}) \subseteq \mathbf{S}'$ *and* $\mathbf{X} \cap \mathbf{S}' = \emptyset$*.*

7 Conclusion

We have introduced hybridizations, and used them to prove a powerful decomposition theorem for Petri sets. For VAS reachabillity sets the decomposition can be effectively computed. We have derived several geometric and computational results. We think that our decomposition can help to study the computational power of VAS. For example, it leads to this corollary:

▶ **Corollary 7.1.** *Let* $f\colon \mathbb{N} \to \mathbb{N}$ *be a function whose graph does not contain an infinite line. Then either* $\{(x,y) \mid y < f(x)\}$ *or* $\{(x,y) \mid y > f(x)\}$ *is not a Petri set.*

Proof. Assume for contradiction that both are Petri sets. Then, since finite unions of Petri sets are again Petri sets, $\{(x, y) \mid y \neq f(x)\}$ is a Petri set. Its complement is the graph of f, which by assumption does not contain an infinite line. Contradiction to Corollary 6.4. ◀

We plan to study other possible applications of our result, derived from the fact that the reachability relation of a VAS is also a Petri set.

References

1 Dmitry Chistikov and Christoph Haase. The Taming of the Semi-Linear Set. In *ICALP*, volume 55 of *LIPIcs*, pages 128:1–128:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

2 Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The Reachability Problem for Petri Nets is not Elementary. In *STOC*, pages 24–33. ACM, 2019.

3 Wojciech Czerwinski and Lukasz Orlikowski. Reachability in Vector Addition Systems is Ackermann-complete. In *FOCS*, pages 1229–1240. IEEE, 2021.

4 Seymour Ginsburg and Edwin H Spanier. *Bounded ALGOL-like Languages*. SDC, 1963.

5 Roland Guttenberg, Mikhail Raskin, and Javier Esparza. Geometry of Reachability Sets of Vector Addition Systems, 2023. `arXiv:2211.02889`.

6 Dirk Hauschildt. *Semilinearity of the Reachability Set is decidable for Petri Nets*. PhD thesis, University of Hamburg, Germany, 1990.

7 Petr Jancar, Jérôme Leroux, and Grégoire Sutre. Co-Finiteness and Co-Emptiness of Reachability Sets in Vector Addition Systems with States. *Fundam. Informaticae*, 169(1-2):123–150, 2019.

8 S. Rao Kosaraju. Decidability of Reachability in Vector Addition Systems. In *STOC*, pages 267–281. ACM, 1982.

9 Jean-Luc Lambert. A Structure to Decide Reachability in Petri Nets. *Theor. Comput. Sci.*, 99(1):79–104, 1992.

10 Jérôme Leroux. The General Vector Addition System Reachability Problem by Presburger Inductive Invariants. In *LICS*, pages 4–13. IEEE Computer Society, 2009.

11 Jérôme Leroux. Vector Addition System Reachability Problem: A Short Self-Contained Proof. In *LATA*, volume 6638 of *Lecture Notes in Computer Science*, pages 41–64. Springer, 2011.

12 Jérôme Leroux. Vector Addition Systems Reachability Problem (A Simpler Solution). In *Turing-100*, volume 10 of *EPiC Series in Computing*, pages 214–228. EasyChair, 2012.

13 Jérôme Leroux. Presburger Vector Addition Systems. In *LICS*, pages 23–32. IEEE Computer Society, 2013. URL: `https://hal.science/hal-00780462v2`.

14 Jérôme Leroux. The Reachability Problem for Petri Nets is not Primitive Recursive. In *FOCS*, pages 1241–1252. IEEE, 2021.

15 Jérôme Leroux and Sylvain Schmitz. Demystifying Reachability in Vector Addition Systems. In *LICS*, pages 56–67. IEEE Computer Society, 2015.

16 Jérôme Leroux and Sylvain Schmitz. Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In *LICS*, pages 1–13. IEEE, 2019.

17 Ernst W. Mayr. An Algorithm for the General Petri Net Reachability Problem. In *STOC*, pages 238–246. ACM, 1981.

18 Danny Nguyen and Igor Pak. Enumerating Projections of Integer Points in Unbounded Polyhedra. *SIAM J. Discret. Math.*, 32(2):986–1002, 2018.

19 Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1999.

20 Kevin Woods. Presburger Arithmetic, Rational Generating Functions, and Quasi-Polynomials. *J. Symb. Log.*, 80(2):433–449, 2015.

Safety Analysis of Parameterised Networks with Non-Blocking Rendez-Vous

Lucie Guillou
IRIF, CNRS, Université Paris Cité, France

Arnaud Sangnier
IRIF, CNRS, Université Paris Cité, France

Nathalie Sznajder
LIP6, CNRS, Sorbonne Université, France

Abstract

We consider networks of processes that all execute the same finite-state protocol and communicate via a rendez-vous mechanism. When a process requests a rendez-vous, another process can respond to it and they both change their control states accordingly. We focus here on a specific semantics, called non-blocking, where the process requesting a rendez-vous can change its state even if no process can respond to it. In this context, we study the parameterised coverability problem of a configuration, which consists in determining whether there is an initial number of processes and an execution allowing to reach a configuration bigger than a given one. We show that this problem is EXPSPACE-complete and can be solved in polynomial time if the protocol is partitioned into two sets of states, the states from which a process can request a rendez-vous and the ones from which it can answer one. We also prove that the problem of the existence of an execution bringing all the processes in a final state is undecidable in our context. These two problems can be solved in polynomial time with the classical rendez-vous semantics.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Parameterised verification, Coverability, Counter machines

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.7

Related Version *Full Version*: `https://arxiv.org/abs/2307.04546` [14]

1 Introduction

Verification of distributed/concurrent systems. Because of their ubiquitous use in applications we rely on constantly, the development of formal methods to guarantee the correct behaviour of distributed/concurrent systems has become one of the most important research directions in the field of computer systems verification in the last two decades. Unfortunately, such systems are difficult to analyse for several reasons. Among others, we can highlight two aspects that make the verification process tedious. First, these systems often generate a large number of different executions due to the various interleavings generated by the concurrent behaviours of the entities involved. Understanding how these interleavings interact is a complex task which can often lead to errors at the design-level or make the model of these systems very complex. Second, in some cases, the number of participants in a distributed system may be unbounded and not known a priori. To fully guarantee the correctness of such systems, the analysis would have to be performed for all possible instances of the system, i.e., an infinite number of times. As a consequence, classical techniques to verify finite state systems, like testing or model-checking, cannot be easily adapted to distributed systems and it is often necessary to develop new techniques.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 7; pp. 7:1–7:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Parameterised verification. When designing systems with an unbounded number of participants, one often provides a schematic program (or protocol) intended to be implemented by multiple identical processes, parameterised by the number of participants. In general, even if the verification problem is decidable for a given instance of the parameter, verifying all possible instances is undecidable ([3]). However, several settings come into play that can be adjusted to allow automatic verification. One key aspect to obtain decidability is to assume that the processes do not manipulate identities and use simple communication mechanisms like pairwise synchronisation (or rendez-vous) [13], broadcast of a message to all the entities [10] (which can as well be lossy in order to simulate mobility [6]), shared register containing values of a finite set [11], and so on (see [9] for a survey). In every aforementioned case, all the entities execute the same protocol given by a finite state automaton. Note that parameterised verification, when decidable like in the above models, is also sometimes surprisingly easy, compared to the same problem with a fixed number of participants. For instance, liveness verification of parameterised systems with shared memory is PSPACE-complete for a fixed number of processes and in NP when parameterised [7].

Considering rendez-vous communication. In one of the seminal papers for the verification of parameterised networks [13], German and Sistla (and since then [4, 15]) assume that the entities communicate by "rendez-vous", a synchronisation mechanism in which two processes (the *sender* and the *receiver*) agree on a common action by which they jointly change their local state. This mechanism is synchronous and symmetric, meaning that if no process is ready to receive a message, the sender cannot send it. However, in some applications, such as Java Thread programming, this is not exactly the primitive that is implemented. When a Thread is suspended in a waiting state, it is woken up by the reception of a message `notify` sent by another Thread. However, the sender is not blocked if there is no suspended Thread waiting for its message; in this case, the sender sends the `notify` anyway and the message is simply lost. This is the reason why Delzanno et. al. have introduced *non-blocking* rendez-vous in [5] a communication primitive in which the sender of a message is not blocked if no process receives it. One of the problems of interest in parameterised verification is the coverability problem: is it possible that, starting from an initial configuration, (at least) one process reaches a bad state? In [5], and later in [20], the authors introduce variants of Petri nets to handle this type of communication. In particular, the authors investigate in [20] the coverability problem for an extended class of Petri nets with non-blocking arcs, and show that for this model the coverability problem is decidable using the techniques of Well-Structured Transitions Systems [1, 2, 12]. However, since their model is an extension of Petri nets, the latter problem is EXPSPACE-hard [17] (no upper bound is given). Relying on Petri nets to obtain algorithms for parameterised networks is not always a good option. In fact, the coverability problem for parameterised networks with rendez-vous is in P [13], while it is EXPSPACE-complete for Petri nets [19, 17]. Hence, no upper bound or lower bound can be directly deduced for the verification of networks with non-blocking rendez-vous from [20].

Our contributions. We show that the coverability problem for parameterised networks with *non-blocking rendez-vous communication* over a finite alphabet is EXPSPACE-complete. To obtain this result, we consider an extension of counter machines (without zero test) where we add non-blocking decrement actions and edges that can bring back the machine to its initial location at any moment. We show that the coverability problem for these extended counter machines is EXPSPACE-complete (Section 3) and that it is equivalent to our problem over parameterised networks (Section 4). We consider then a subclass of parameterised networks – *wait-only protocols* – in which no state can allow to both request a rendez-vous and wait for one. This restriction is very natural to model concurrent programs since when a thread is waiting, it cannot perform any other action. We show that coverability problem can then be solved in polynomial time (Section 5). Finally, we show that the synchronization problem, where we look for a reachable configuration with all the processes in a given state, is undecidable in our framework, even for wait-only protocols (Section 6).

Due to lack of space, some proofs are only given in [14].

2 Rendez-vous Networks with Non-Blocking Semantics

For a finite alphabet Σ, we let Σ^* denote the set of finite sequences over Σ (or words). Given $w \in \Sigma^*$, we let $|w|$ denote its length: if $w = w_0 \dots w_{n-1} \in \Sigma^*$, then $|w| = n$. We write $\mathbb{N}$ to denote the set of natural numbers and $[i, j]$ to represent the set $\{k \in \mathbb{N} \mid i \leq k \text{ and } k \leq j\}$ for $i, j \in \mathbb{N}$. For a finite set E, the set $\mathbb{N}^E$ represents the multisets over E. For two elements $m, m' \in \mathbb{N}^E$, we denote $m + m'$ the multiset such that $(m + m')(e) = m(e) + m'(e)$ for all $e \in E$. We say that $m \leq m'$ if and only if $m(e) \leq m'(e)$ for all $e \in E$. If $m \leq m'$, then $m' - m$ is the multiset such that $(m' - m)(e) = m'(e) - m(e)$ for all $e \in E$. Given a subset $E' \subseteq E$ and $m \in \mathbb{N}^E$, we denote by $||m||_{E'}$ the sum $\Sigma_{e \in E'} m(e)$ of elements of E' present in m. The size of a multiset m is given by $||m|| = ||m||_E$. For $e \in E$, we use sometimes the notation $\wr e \int$ for the multiset m verifying $m(e) = 1$ and $m(e') = 0$ for all $e' \in E \setminus \{e\}$ and, to represent for instance the multiset with four elements a, b, b and c, we will also use the notations $\wr a, b, b, c \int$ or $\wr a, 2 \cdot b, c \int$.

2.1 Rendez-Vous Protocols

We can now define our model of networks. We assume that all processes in the network follow the same protocol. Communication in the network is pairwise and is performed by *rendez-vous* through a finite communication alphabet Σ. Each process can either perform an internal action using the primitive τ, or request a rendez-vous by sending the message m using the primitive $!m$ or answer to a rendez-vous by receiving the message m using the primitive $?m$ (for $m \in \Sigma$). Thus, the set of primitives used by our protocols is $RV(\Sigma) = \{\tau\} \cup \{?m, !m \mid m \in \Sigma\}$.

▶ **Definition 2.1** (Rendez-vous protocol). *A* rendez-vous protocol *(shortly protocol) is a tuple $\mathcal{P} = (Q, \Sigma, q_{in}, q_f, T)$ where Q is a finite set of states, Σ is a finite alphabet, $q_{in} \in Q$ is the initial state, $q_f \in Q$ is the final state and $T \subseteq Q \times RV(\Sigma) \times Q$ is the finite set of transitions.*

For a message $m \in \Sigma$, we denote by $R(m)$ the set of states q from which the message m can be received, i.e. states q such that there is a transition $(q, ?m, q') \in T$ for some $q' \in Q$.

A *configuration* associated to the protocol $\mathcal{P}$ is a non-empty multiset C over Q for which $C(q)$ denotes the number of processes in the state q and $||C||$ denotes the total number of processes in the configuration C. A configuration C is said to be *initial* if and only if $C(q) = 0$ for all $q \in Q \setminus \{q_{in}\}$. We denote by $\mathcal{C}(\mathcal{P})$ the set of configurations and by $\mathcal{I}(\mathcal{P})$ the set of initial configurations. Finally for $n \in \mathbb{N} \setminus \{0\}$, we use the notation $\mathcal{C}_n(\mathcal{P})$ to represent the set of configurations of size n, i.e. $\mathcal{C}_n(\mathcal{P}) = \{C \in \mathcal{C}(\mathcal{P}) \mid ||C|| = n\}$. When the protocol is made clear from the context, we shall write $\mathcal{C}$, $\mathcal{I}$ and $\mathcal{C}_n$.

We explain now the semantics associated with a protocol. For this matter we define the relation $\rightarrow_{\mathcal{P}} \subseteq \bigcup_{n \geq 1} \mathcal{C}_n \times \big(\{\tau\} \cup \Sigma \cup \{\mathbf{nb}(m) \mid m \in \Sigma\}\big) \times \mathcal{C}_n$ as follows (here $\mathbf{nb}(\cdot)$ is a special symbol). Given $n \in \mathbb{N} \setminus \{0\}$ and $C, C' \in \mathcal{C}_n$ and $m \in \Sigma$, we have:

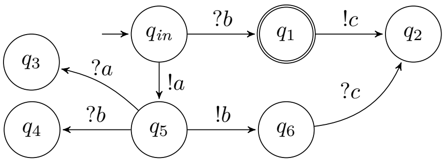


Figure 1 Example of a rendez-vous protocol $\mathcal{P}$.

1. $C \xrightarrow{\tau}_{\mathcal{P}} C'$ iff there exists $(q, \tau, q') \in T$ such that $C(q) > 0$ and $C' = C - \wr q \wr + \wr q' \wr$ **(internal)**;
2. $C \xrightarrow{m}_{\mathcal{P}} C'$ iff there exists $(q_1, !m, q_1') \in T$ and $(q_2, ?m, q_2') \in T$ such that $C(q_1) > 0$ and $C(q_2) > 0$ and $C(q_1) + C(q_2) \geq 2$ (needed when $q_1 = q_2$) and $C' = C - \wr q_1, q_2 \wr + \wr q_1', q_2' \wr$ **(rendez-vous)**;
3. $C \xrightarrow{\mathbf{nb}(m)}_{\mathcal{P}} C'$ iff there exists $(q_1, !m, q_1') \in T$, such that $C(q_1) > 0$ and $(C - \wr q_1 \wr)(q_2) = 0$ for all $(q_2, ?m, q_2') \in T$ and $C' = C - \wr q_1 \wr + \wr q_1' \wr$ **(non-blocking request)**.

Intuitively, from a configuration C, we allow the following behaviours: either a process takes an internal transition (labeled by τ), or two processes synchronize over a rendez-vous m, or a process requests a rendez-vous to which no process can answer (non-blocking sending).

This allows us to define $S_{\mathcal{P}}$ the transition system $(\mathcal{C}(\mathcal{P}), \rightarrow_{\mathcal{P}})$ associated to $\mathcal{P}$. We will write $C \rightarrow_{\mathcal{P}} C'$ when there exists $a \in \{\tau\} \cup \Sigma \cup \{\mathbf{nb}(m) \mid m \in \Sigma\}$ such that $C \xrightarrow{a}_{\mathcal{P}} C'$ and denote by $\rightarrow_{\mathcal{P}}^*$ the reflexive and transitive closure of $\rightarrow_{\mathcal{P}}$. Furthermore, when made clear from the context, we might simply write $\rightarrow$ instead of $\rightarrow_{\mathcal{P}}$. An *execution* is a finite sequence of configurations $\rho = C_0 C_1 \ldots$ such that, for all $0 \leq i < |\rho|$, $C_i \rightarrow_{\mathcal{P}} C_{i+1}$. The execution is said to be initial if $C_0 \in \mathcal{I}(\mathcal{P})$.

▶ **Example 2.2.** Figure 1 provides an example of a rendez-vous protocol where q_{in} is the initial state and q_1 the final state. A configuration associated to this protocol is for instance the multiset $\wr 2 \cdot q_1, 1 \cdot q_4, 1 \cdot q_5 \wr$ and the following sequence represents an initial execution: $\wr 2 \cdot q_{in} \wr \xrightarrow{\mathbf{nb}(a)} \wr q_{in}, q_5 \wr \xrightarrow{b} \wr q_1, q_6 \wr \xrightarrow{c} \wr 2 \cdot q_2 \wr$.

▶ Remark 2.3. When we only allow behaviours of type **(internal)** and **(rendez-vous)**, this semantics corresponds to the classical rendez-vous semantics ([13, 4, 15]). In opposition, we will refer to the semantics defined here as the *non-blocking semantics* where a process is not *blocked* if it requests a rendez-vous and no process can answer to it. Note that all behaviours possible in the classical rendez-vous semantics are as well possible in the non-blocking semantics but the converse is false.

2.2 Verification Problems

We now present the problems studied in this work. For this matter, given a protocol $\mathcal{P} = (Q, \Sigma, q_{in}, q_f, T)$, we define two sets of final configurations. The first one $\mathcal{F}_{\exists}(\mathcal{P}) = \{C \in \mathcal{C}(\mathcal{P}) \mid C(q_f) > 0\}$ characterises the configurations where one of the processes is in the final state. The second one $\mathcal{F}_{\forall}(\mathcal{P}) = \{C \in \mathcal{C}(\mathcal{P}) \mid C(Q \setminus \{q_f\}) = 0\}$ represents the configurations where all the processes are in the final state. Here again, when the protocol is clear from the context, we might use the notations $\mathcal{F}_{\exists}$ and $\mathcal{F}_{\forall}$. We study three problems: the *state coverability problem* (SCover), the *configuration coverability* problem (CCover) and the *synchronization problem* (Synchro), which all take as input a protocol $\mathcal{P}$ and can be stated as follows:

Problem name	**Question**
SCover	Are there $C_0 \in \mathcal{I}$ and $C_f \in \mathcal{F}_\exists$, such that $C_0 \rightarrow^* C_f$?
CCover	Given $C \in \mathcal{C}$, are there $C_0 \in \mathcal{I}$ and $C' \geq C$, such that $C_0 \rightarrow^* C'$?
Synchro	Are there $C_0 \in \mathcal{I}$ and $C_f \in \mathcal{F}_\forall$, such that $C_0 \rightarrow^* C_f$?

▶ Remark 2.4. The difficulty in solving these problems lies in the fact that we are seeking for an initial configuration allowing a specific execution but the set of initial configurations is infinite.

The difference between SCover and Synchro is that in the first one we ask for at least one process to end up in the final state whereas the second one requires all the processes to end in this state. Note that SCover is an instance of CCover but Synchro is not.

Observe that SCover should be seen as a safety property: if q_f is an error state and the answer is negative, then for any number of processes, no process will ever be in that error state.

▶ **Example 2.5.** The rendez-vous protocol of Figure 1 is a positive instance of SCover, as shown in Example 2.2. However, this is not the case for Synchro: if an execution brings a process in q_2, this process cannot be brought afterwards to q_1. If q_2 is the final state, $\mathcal{P}$ is now a positive instance of Synchro (see Example 2.2). Note that if the final state is q_4, $\mathcal{P}$ is not a positive instance of SCover anymore. In fact, the only way to reach a configuration with a process in q_4 is to put (at least) two processes in state q_5 as this is the only state from which one process can send the message b. However, this cannot happen, since from an initial configuration, the only available action consists in sending the message a as a non-blocking request. Once there is one process in state q_5, any other attempt to put another process in this state will induce a reception of message a by the process already in q_5, which will hence leave q_5. Finally, note that for any $n \in \mathbb{N}$, the configuration $\wr n \cdot q_3 \wr$ is coverable, even if $\mathcal{P}$ with q_3 as final state is not a positive instance of Synchro.

3 Coverability for Non-Blocking Counter Machines

We first detour into new classes of counter machines, which we call *non-blocking counter machines* and *non-blocking counter machines with restore*, in which a new way of decrementing the counters is added to the classical one: a non-blocking decrement, which is an action that can always be performed. If the counter is strictly positive, it is decremented; otherwise it is let to 0. We show that the coverability of a control state in this model is Expspace-complete, and use this result to solve coverability problems in rendez-vous protocols.

To define counter machines, given a set of integer variables (also called counters) X, we use the notation $\mathsf{CAct}(X)$ to represent the set of associated actions given by $\{\mathtt{x}+, \mathtt{x}-, \mathtt{x}{=}0 \mid \mathtt{x} \in X\} \cup \{\perp\}$. Intuitively, $\mathtt{x}+$ increments the value of the counter $\mathtt{x}$, while $\mathtt{x}-$ decrements it and $\mathtt{x}{=}0$ checks if it is equal to 0. We are now ready to state the syntax of this model.

▶ **Definition 3.1.** *A* counter machine *(shortly CM) is a tuple $M = (Loc, X, \Delta, \ell_{in})$ such that Loc is a finite set of locations, $\ell_{in} \in Loc$ is an initial location, X is a finite set of counters, and $\Delta \subseteq Loc \times \mathsf{CAct}(X) \times Loc$ is finite set of transitions.*

We will say that a CM is test-free (shortly test-free CM) whenever $\Delta \cap \mathrm{Loc} \times \{\mathtt{x}{=}0 \mid \mathtt{x} \in X\} \times \mathrm{Loc} = \emptyset$. A configuration of a CM $M = (\mathrm{Loc}, X, \Delta, \ell_{in})$ is a pair (ℓ, v) where $\ell \in \mathrm{Loc}$ specifies the current location of the CM and $v \in \mathbb{N}^X$ associates to each counter a natural value. The size of a CM M is given by $|M| = |\mathrm{Loc}| + |X| + |\Delta|$. Given two configurations (ℓ, v) and (ℓ', v') and a transition $\delta \in \Delta$, we define $(\ell, v) \overset{\delta}{\rightsquigarrow}_M (\ell', v')$ if and only if $\delta = (\ell, op, \ell')$ and one of the following holds:

- $op = \bot$ and $v = v'$;
- $op = \mathtt{x}+$ and $v'(\mathtt{x}) = v(\mathtt{x}) + 1$ and $v'(\mathtt{x}') = v(\mathtt{x}')$ for all $\mathtt{x}' \in X \setminus \{\mathtt{x}\}$;
- $op = \mathtt{x}-$ and $v'(\mathtt{x}) = v(\mathtt{x}) - 1$ and $v'(\mathtt{x}') = v(\mathtt{x}')$ for all $\mathtt{x}' \in X \setminus \{\mathtt{x}\}$;
- $op = \mathtt{x}{=}0$ and $v(\mathtt{x}) = 0$ and $v' = v$.

In order to simulate the non-blocking semantics of our rendez-vous protocols with counter machines, we extend the class of test-free CM with non-blocking decrement actions.

▶ **Definition 3.2.** *A* non-blocking test-free counter machine *(shortly* NB-CM*) is a tuple $M = (Loc, X, \Delta_b, \Delta_{nb}, \ell_{in})$ such that $(Loc, X, \Delta_b, \ell_{in})$ is a test-free CM and $\Delta_{nb} \subseteq Loc \times \{nb(\mathtt{x}-) \mid \mathtt{x} \in X\} \times Loc$ is a finite set of* non-blocking *transitions.*

Observe that in a NB-CM, both blocking and non-blocking decrements are possible, depending on the type of transition taken. Again, a configuration is given by a pair $(\ell, v) \in \text{Loc} \times \mathbb{N}^X$. Given two configurations (ℓ, v) and (ℓ, v') and $\delta \in \Delta_b \cup \Delta_{nb}$, we extend the transition relation $(\ell, v) \overset{\delta}{\rightsquigarrow}_M (\ell, v')$ over the set Δ_{nb} in the following way: for $\delta = (\ell, nb(\mathtt{x}-), \ell') \in \Delta_{nb}$, we have $(\ell, v) \overset{\delta}{\rightsquigarrow}_M (\ell', v')$ if and only if $v'(\mathtt{x}) = \max(0, v(\mathtt{x}) - 1)$, and $v'(\mathtt{x}') = v(\mathtt{x}')$ for all $\mathtt{x}' \in X \setminus \{\mathtt{x}\}$.

We say that M is an NB-CM *with restore* (shortly NB-R-CM) when $(\ell, \bot, \ell_{in}) \in \Delta$ for all $\ell \in \text{Loc}$, i.e. from each location, there is a transition leading to the initial location with no effect on the counters values.

For a CM M with set of transitions Δ (resp. an NB-CM with sets of transitions Δ_b and Δ_{nb}), we will write $(\ell, v) \rightsquigarrow_M (\ell', v')$ whenever there exists $\delta \in \Delta$ (resp. $\delta \in \Delta_b \cup \Delta_{nb}$) such that $(\ell, v) \overset{\delta}{\rightsquigarrow}_M (\ell', v')$ and use $\rightsquigarrow^*_M$ to represent the reflexive and transitive closure of $\rightsquigarrow_M$. When the context is clear we shall write $\rightsquigarrow$ instead of $\rightsquigarrow_M$. We let $\mathbf{0}_X$ be the valuation such that $\mathbf{0}_X(\mathtt{x}) = 0$ for all $\mathtt{x} \in X$. An execution is a finite sequence of configurations $(\ell_0, v_0) \rightsquigarrow (\ell_1, v_1) \rightsquigarrow \ldots \rightsquigarrow (\ell_k, v_k)$. It is said to be initial if $(\ell_0, v_0) = (\ell_{in}, \mathbf{0}_X)$. A configuration (ℓ, v) is called reachable if $(\ell_{in}, \mathbf{0}_X) \rightsquigarrow^* (\ell, v)$.

We shall now define the coverability problem for (non-blocking test-free) counter machines, which asks whether a given location can be reached from the initial configuration. We denote this problem COVER[$\mathcal{M}$], for $\mathcal{M} \in \{\text{CM}, \text{test-free CM}, \text{NB-CM}, \text{NB-R-CM}\}$. It takes as input a machine M in $\mathcal{M}$ (with initial location ℓ_{in} and working over a set X of counters) and a location ℓ_f and it checks whether there is a valuation $v \in \mathbb{N}^X$ such that $(\ell_{in}, \mathbf{0}_X) \rightsquigarrow^* (\ell_f, v)$.

In the rest of this section, we will prove that COVER[NB-R-CM] is EXPSPACE-complete. To this end, we first establish that COVER[NB-CM] is in EXPSPACE, by an adaptation of Rackoff's proof which shows that coverability in Vector Addition Systems is in EXPSPACE [19]. This gives also the upper bound for NB-R-CM, since any NB-R-CM is a NB-CM. This result is established by the following theorem, whose proof is omitted due to lack of space.

▶ **Theorem 3.3.** *COVER*[NB-CM] *and COVER*[NB-R-CM] *are in EXPSPACE.*

To obtain the lower bound, inspired by Lipton's proof showing that coverability in Vector Addition Systems is EXPSPACE-hard [8, 17], we rely on 2EXP-bounded test-free CM. We say that a CM $M = (\text{Loc}, X, \Delta, \ell_{in})$ is 2EXP-*bounded* if there exists $n \in O(|M|)$ such that any reachable configuration (ℓ, v) satisfies $v(\mathtt{x}) \leq 2^{2^n}$ for all $\mathtt{x} \in X$. We use then the following result.

▶ **Theorem 3.4** ([8, 17]). *COVER[2EXP-bounded test-free CM] is EXPSPACE-hard.*

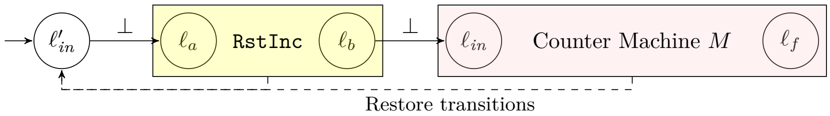


Figure 2 The NB-R-CM N.

We now show how to simulate a 2Exp-bounded test-free CM by a NB-R-CM, by carefully handling restore transitions that may occur at any point in the execution. We will ensure that each restore transition is followed by a reset of the counters, so that we can always extract from an execution of the NB-R-CM a correct initial execution of the original test-free CM. The way we enforce resetting of the counters is inspired by the way Lipton simulates 0-tests of a CM in a test-free CM. As in [17, 8], we will describe the final NB-R-CM by means of several submachines. To this end, we define *procedural non-blocking counter machines* that are NB-CM with several identified *output states*: formally, a procedural-NB-CM is a tuple $N = (\mathrm{Loc}, X, \Delta_b, \Delta_{nb}, \ell_{in}, L_{out})$ such that $(\mathrm{Loc}, X, \Delta_b, \Delta_{nb}, \ell_{in})$ is a NB-CM, $L_{out} \subseteq \mathrm{Loc}$, and there is no outgoing transition from states in L_{out}.

Now fix a 2Exp-bounded test-free CM $M = (\mathrm{Loc}, X, \Delta, \ell_{in})$, $\ell_f \in \mathrm{Loc}$ the location to be covered. There is some c, such that, any reachable configuration (ℓ, v) satisfies $v(\mathtt{x}) < 2^{2^{c|M|}}$ for all $\mathtt{x} \in X$, fix $n = c|M|$. We build a NB-R-CM N as pictured in Figure 2. The goal of the procedural NB-CM $\mathtt{RstInc}$ is to ensure that all counters in X are reset. Hence, after each restore transition, we are sure that we start over a fresh execution of the test-free CM M. We will need the mechanism designed by Lipton to test whether a counter is equal to 0. So, we define two families of sets of counters $(Y_i)_{0\le i\le n}$ and $(\overline{Y_i})_{0\le i\le n}$ as follows. Let $Y_i = \{\mathtt{y}_i, \mathtt{z}_i, \mathtt{s}_i\}$ and $\overline{Y}_i = \{\overline{\mathtt{y}}_i, \overline{\mathtt{z}}_i, \overline{\mathtt{s}}_i\}$ for all $0 \le i < n$ and $Y_n = X$ and $\overline{Y}_n = \emptyset$ and $X' = \bigcup_{0\le i\le n} Y_i \cup \overline{Y}_i$. All the machines we will describe from now on will work over the set of counters X'.

Procedural-NB-CM $\mathtt{TestSwap}_i(\mathtt{x})$. We use a family of procedural-NB-CM defined in [17, 8]: for all $0 \le i < n$, for all $\overline{\mathtt{x}} \in \overline{Y}_i$, $\mathtt{TestSwap}_i(\overline{\mathtt{x}})$ is a procedural-NB-CM with an initial location $\ell_{in}^{\mathtt{TS},i,\mathtt{x}}$, and two output locations $\ell_z^{\mathtt{TS},i,\mathtt{x}}$ and $\ell_{nz}^{\mathtt{TS},i,\mathtt{x}}$. It tests if the value of $\overline{\mathtt{x}}$ is equal to 0, using the fact that the sum of the values of $\mathtt{x}$ and $\overline{\mathtt{x}}$ is equal to 2^{2^i}. If $\overline{\mathtt{x}} = 0$, it swaps the values of $\mathtt{x}$ and $\overline{\mathtt{x}}$, and the execution ends in the output location $\ell_z^{\mathtt{TS},i,\mathtt{x}}$. Otherwise, counters values are left unchanged and the execution ends in $\ell_{nz}^{\mathtt{TS},i,\mathtt{x}}$. In any case, other counters are not modified by the execution. Note that $\mathtt{TestSwap}_i(\mathtt{x})$ makes use of variables in $\bigcup_{1\le j<i} Y_i \cup \overline{Y}_i$.

Procedural NB-CM $\mathtt{Rst}_i$. We use these machines to define a family of procedural-NB-CM called $(\mathtt{Rst}_i)_{0\le i\le n}$ that reset the counters in $Y_i \cup \overline{Y_i}$, assuming that their values are less than or equal to 2^{2^i}. Let $0 \le i \le n$, we let $\mathtt{Rst}_i = (\mathrm{Loc}^{\mathtt{R},i}, X', \Delta_b^{\mathtt{R},i}, \Delta_{nb}^{\mathtt{R},i}, \ell_{in}^{\mathtt{R},i}, \{\ell_{out}^{\mathtt{R},i}\})$. The machine $\mathtt{Rst}_0$ is pictured Figure 3. For all $0 \le i < n$, the machine $\mathtt{Rst}_{i+1}$ uses counters from $Y_i \cup \overline{Y_i}$ and procedural-NB-CM $\mathtt{Testswap}_i(\overline{\mathtt{z}}_i)$ and $\mathtt{Testswap}_i(\overline{\mathtt{y}}_i)$ to control the number of times variables from Y_{i+1} and $\overline{Y}_{i+1}$ are decremented. It is pictured Figure 4. Observe that since $Y_n = X$, and $\overline{Y_n} = \emptyset$, the machine $\mathtt{Rst}_n$ will be a bit different from the picture: there will only be non-blocking decrements over counters from Y_n, that is over counters X from the initial test-free CM M. If $\overline{\mathtt{y}}_i$, $\overline{\mathtt{z}}_i$ (and $\overline{\mathtt{s}}_i$) are set to 2^{2^i} and $\mathtt{y}_i$, $\mathtt{z}_i$ (and $\mathtt{s}_i$) are set to 0, then each time this procedural-NB-CM takes an outer loop, the variables of $Y_{i+1} \cup \overline{Y}_{i+1}$ are decremented (in a non-blocking fashion) 2^{2^i} times. This is ensured by the properties of $\mathtt{TestSwap}_i(\mathtt{x})$. Moreover, the location $\ell_z^{\mathtt{TS},i,\mathtt{y}}$ will only be reached when the counter $\overline{\mathtt{y}}_i$

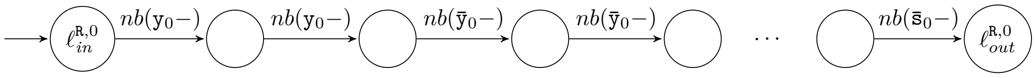

$\ell_{in}^{\mathtt{R},0}$
$nb(\mathtt{y}_0-)$
$nb(\mathtt{y}_0-)$
$nb(\overline{\mathtt{y}}_0-)$
$nb(\overline{\mathtt{y}}_0-)$
$\cdots$
$nb(\overline{\mathtt{s}}_0-)$
$\ell_{out}^{\mathtt{R},0}$

Figure 3 Description of $\mathtt{Rst}_0$.

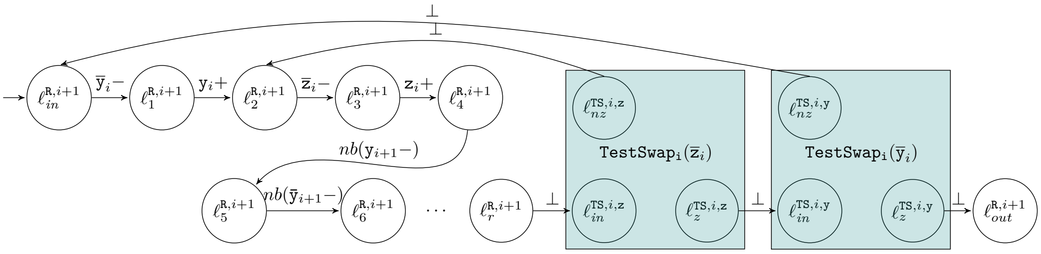

$\perp$
$\perp$
$\ell_{in}^{\mathtt{R},i+1}$
$\overline{\mathtt{y}}_i-$
$\ell_1^{\mathtt{R},i+1}$
$\mathtt{y}_i+$
$\ell_2^{\mathtt{R},i+1}$
$\overline{\mathtt{z}}_i-$
$\ell_3^{\mathtt{R},i+1}$
$\mathtt{z}_i+$
$\ell_4^{\mathtt{R},i+1}$
$\ell_{nz}^{\mathtt{TS},i,\mathtt{z}}$
$\mathtt{TestSwap_i}(\overline{\mathtt{z}}_i)$
$\ell_{nz}^{\mathtt{TS},i,\mathtt{y}}$
$\mathtt{TestSwap_i}(\overline{\mathtt{y}}_i)$
$nb(\mathtt{y}_{i+1}-)$
$\ell_5^{\mathtt{R},i+1}$
$nb(\overline{\mathtt{y}}_{i+1}-)$
$\ell_6^{\mathtt{R},i+1}$
$\cdots$
$\ell_r^{\mathtt{R},i+1}$
$\perp$
$\ell_{in}^{\mathtt{TS},i,\mathtt{z}}$
$\ell_z^{\mathtt{TS},i,\mathtt{z}}$
$\perp$
$\ell_{in}^{\mathtt{TS},i,\mathtt{y}}$
$\ell_z^{\mathtt{TS},i,\mathtt{y}}$
$\perp$
$\ell_{out}^{\mathtt{R},i+1}$

Figure 4 Description of $\mathtt{Rst_{i+1}}$.

is set to 0, and this will happen after 2^{2^i} iterations of the outer loop, again thanks to the properties of $\mathtt{TestSwap}_i(\mathtt{x})$. So, all in all, variables from Y_i and $\overline{Y}_{i+1}$ will take a non-blocking decrement $2^{2^i}.2^{2^i}$ times, that is $2^{2^{i+1}}$.

For all $\mathtt{x} \in X'$, we say that $\mathtt{x}$ is *initialized* in a valuation v if $\mathtt{x} \in Y_i$ for some $0 \leq i \leq n$ and $v(\mathtt{x}) = 0$, or $\mathtt{x} \in \overline{Y}_i$ for some $0 \leq i \leq n$ and $v(\mathtt{x}) = 2^{2^i}$. For $0 \leq i \leq n$, we say that a valuation $v \in \mathbb{N}^{X'}$ is *i-bounded* if for all $\mathtt{x} \in Y_i \cup \overline{Y}_i$, $v(\mathtt{x}) \leq 2^{2^i}$.

The construction ensures that when one enters $\mathtt{Rst}_i$ with a valuation v that is i-bounded, and in which all variables in $\bigcup_{0 \leq j < i} Y_j \cup \overline{Y}_j$ are initialized, the location $\ell_{out}^{\mathtt{R},i}$ is reached with a valuation v' such that: $v'(\mathtt{x}) = 0$ for all $\mathtt{x} \in Y_i \cup \overline{Y}_i$ and $v'(\mathtt{x}) = v(\mathtt{x})$ for all $\mathtt{x} \notin Y_i \cup \overline{Y}_i$. Moreover, if v is j-bounded for all $0 \leq j \leq n$, then any valuation reached during the execution remains j-bounded for all $0 \leq j \leq n$.

Procedural NB-CM $\mathtt{Inc}_i$**.** The properties we seek for $\mathtt{Rst}_i$ are ensured whenever the variables in $\bigcup_{0 \leq j < i} Y_j \cup \overline{Y}_j$ are initialized. This is taken care of by a family of procedural-NB-CM introduced in [17, 8]. For all $0 \leq i < n$, $\mathtt{Inc}_i$ is a procedural-NB-CM with initial location $\ell_{in}^{\mathtt{Inc},i}$, and unique output location $\ell_{out}^{\mathtt{Inc},i}$. They enjoy the following property: for $0 \leq i < n$, when one enters $\mathtt{Inc}_i$ with a valuation v in which all the variables in $\bigcup_{0 \leq j < i} Y_j \cup \overline{Y}_j$ are initialized and $v(\mathtt{x}) = 0$ for all $\mathtt{x} \in \overline{Y}_i$, then the location $\ell_{out}^{\mathtt{Inc}_i}$ is reached with a valuation v' such that $v'(\mathtt{x}) = 2^{2^i}$ for all $\mathtt{x} \in \overline{Y}_i$, and $v'(\mathtt{x}) = v(\mathtt{x})$ for all other $\mathtt{x} \in X'$. Moreover, if v is j-bounded for all $0 \leq j \leq n$, then any valuation reached during the execution remains j-bounded for all $0 \leq j \leq n$.

Procedural NB-CM $\mathtt{RstInc}$**.** Finally, let $\mathtt{RstInc}$ be a procedural-NB-CM with initial location ℓ_a and output location ℓ_b, over the set of counters X' and built as an alternation of $\mathtt{Rst}_i$ and $\mathtt{Inc}_i$ for $0 \leq i < n$, finished by $\mathtt{Rst}_n$. It is depicted in Figure 5. Thanks to the properties of the machines $\mathtt{Rst}_i$ and $\mathtt{Inc}_i$, in the output location of each $\mathtt{Inc}_i$ machine, the counters in $\overline{Y}_i$ are set to 2^{2^i}, which allow counters in $Y_{i+1} \cup \overline{Y}_{i+1}$ to be set to 0 in the output location of $\mathtt{Rst}_{i+1}$. Hence, in location $\ell_{out}^{\mathtt{Inc},n}$, counters in $Y_n = X$ are set to 0.

From [17, 8], each procedural machine $\mathtt{TestSwap}_i(\mathtt{x})$ and $\mathtt{Inc}_i$ has size at most $C \times n^2$ for some constant C. Hence, observe that N is of size at most B for some $B \in O(|M|^3)$. One can show that $(\ell_{in}, \mathbf{0}_X) \rightsquigarrow_M^* (\ell_f, v)$ for some $v \in \mathbb{N}^X$, if and only if $(\ell'_{in}, \mathbf{0}_{X'}) \rightsquigarrow_N^* (\ell_f, v')$ for some $v' \in \mathbb{N}^{X'}$. Using Theorem 3.4, we obtain:

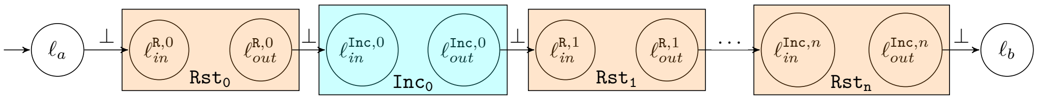

ℓ_a
$\perp$
$\ell_{in}^{\mathrm{R},0}$
$\ell_{out}^{\mathrm{R},0}$
$\mathtt{Rst}_0$
$\perp$
$\ell_{in}^{\mathtt{Inc},0}$
$\ell_{out}^{\mathtt{Inc},0}$
$\mathtt{Inc}_0$
$\perp$
$\ell_{in}^{\mathrm{R},1}$
$\ell_{out}^{\mathrm{R},1}$
$\mathtt{Rst}_1$
$\cdots$
$\ell_{in}^{\mathtt{Inc},n}$
$\ell_{out}^{\mathtt{Inc},n}$
$\mathtt{Rst}_n$
$\perp$
ℓ_b

Figure 5 `RstInc`.

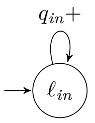


Figure 6 Incrementing q_{in}.

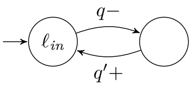


Figure 7 Transitions for $(q, \tau, q') \in T$.

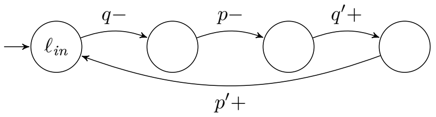


Figure 8 Transitions for a rendez-vous $(q, !a, q')$, $(p, ?a, p') \in T$.

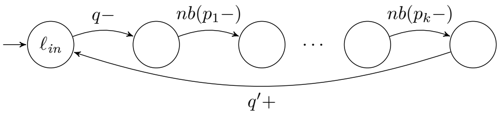


Figure 9 Transitions for a non-blocking sending $(q, !a, q') \in T$ and $R(a) = \{p_1 \dots p_k\}$.

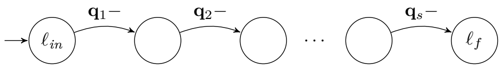


Figure 10 Verification for the coverability of $C_F = \lbag \mathbf{q}_1 \rbag + \lbag \mathbf{q}_2 \rbag + \cdots + \lbag \mathbf{q}_s \rbag$.

▶ **Theorem 3.5.** *Cover[*NB-R-CM*] is Expspace-hard.*

4 Coverability for Rendez-Vous Protocols

In this section we prove that SCover and CCover problems are both Expspace-complete for rendez-vous protocols. To this end, we present the following reductions: CCover reduces to Cover[NB-CM] and Cover[NB-R-CM] reduces to SCover. This will prove that CCover is in Expspace and SCover is Expspace-hard (from Theorem 3.3 and Theorem 3.5). As SCover is an instance of CCover, the two reductions suffice to prove Expspace-completeness for both problems.

4.1 From Rendez-vous Protocols to NB-CM

Let $\mathcal{P} = (Q, \Sigma, q_{in}, q_f, T)$ a rendez-vous protocol and C_F a configuration of $\mathcal{P}$ to be covered. We shall also decompose C_F as a sum of multisets $\lbag \mathbf{q}_1 \rbag + \lbag \mathbf{q}_2 \rbag + \cdots + \lbag \mathbf{q}_s \rbag$. Observe that there might be $\mathbf{q}_i = \mathbf{q}_j$ for $i \neq j$. We build the NB-CM $M = (\mathrm{Loc}, X, \Delta_b, \Delta_{nb}, \ell_{in})$ with $X = Q$. A configuration C of $\mathcal{P}$ is meant to be represented in M by (ℓ_{in}, v), with $v(q) = C(q)$ for all $q \in Q$. The only meaningful location of M is then ℓ_{in}. The other ones are here to ensure correct updates of the counters when simulating a transition. We let $\mathrm{Loc} = \{\ell_{in}\} \cup \{\ell^1_{(t,t')}, \ell^2_{(t,t')}, \ell^3_{(t,t')} \mid t = (q, !a, q'), t' = (p, ?a, p') \in T\} \cup \{\ell_t, \ell^a_{t,p_1}, \cdots, \ell^a_{t,p_k} \mid t = (q, !a, q') \in T, R(a) = \{p_1, \dots, p_k\}\} \cup \{\ell_q \mid t = (q, \tau, q') \in T\} \cup \{\ell_1 \dots \ell_s\}$, with final location $\ell_f = \ell_s$, where $R(m)$ for a message $m \in \Sigma$ has been defined in Section 2. The sets Δ_b and Δ_{nb} are shown Figures 6–10. Transitions pictured Figures 6–8 and 10 show how to simulate a rendez-vous protocol with the classical rendez-vous mechanism. The non-blocking rendez-vous are handled by the transitions pictured Figure 9. If the NB-CM M faithfully simulates $\mathcal{P}$, then this loop of non-blocking decrements is taken when the values of the counters in $R(a)$ are equal to 0, and the configuration reached still corresponds to a configuration in $\mathcal{P}$. However, it could be that this loop is taken in M while some counters in $R(a)$ are strictly positive. In this case, a blocking rendez-vous has to be taken in $\mathcal{P}$, e.g. $(q, !a, q')$ and $(p, ?a, p')$ if the counter p in M is strictly positive. Therefore, the value of the

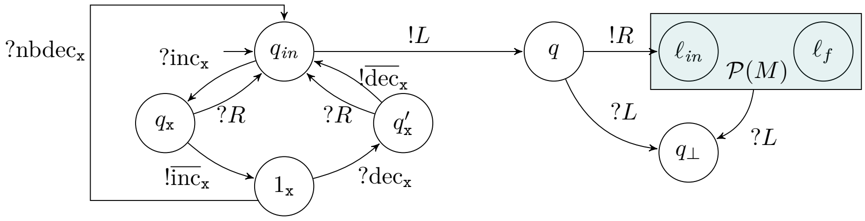


Figure 11 The rendez-vous protocol $\mathcal{P}$ built from the NB-R-CM M. Note that there is one gadget with states $\{q_{\mathtt{x}}, q'_{\mathtt{x}}, 1_{\mathtt{x}}\}$ for each counter $\mathtt{x} \in X$.

reached configuration (ℓ_{in}, v) and the corresponding configuration C in $\mathcal{P}$ will be different: first, $C(p') > v(p')$, since the process in p has moved in the state p' in $\mathcal{P}$ when there has been no increment of p' in M. Furthermore, all other non-blocking decrements of counters in $R(a)$ in M may have effectively decremented the counters, when in $\mathcal{P}$ no other process has left a state of $R(a)$. However, this ensures that $C \geq v$. The reduction then guarantees that if (ℓ_{in}, v) is reachable in M, then a configuration $C \geq v$ is reachable in $\mathcal{P}$. Then, if it is possible to reach a configuration (ℓ_{in}, v) in M whose counters are high enough to cover ℓ_F, then the corresponding initial execution in $\mathcal{P}$ will reach a configuration $C \geq v$, which hence covers C_F.

▶ **Theorem 4.1.** *CCover over rendez-vous protocols is in Expspace.*

4.2 From NB-R-CM to Rendez-Vous Protocols

The reduction from Cover[NB-R-CM] to SCover in rendez-vous protocols mainly relies on the mechanism that can ensure that at most one process evolves in some given set of states, as explained in Example 2.5. This will allow to somehow select a "leader" among the processes that will simulate the behaviour of the NB-R-CM whereas other processes will simulate the values of the counters. Let $M = (\mathrm{Loc}, X, \Delta_b, \Delta_{nb}, \ell_{in})$ a NB-R-CM and $\ell_f \in \mathrm{Loc}$ a final target location. We build the rendez-vous protocol $\mathcal{P}$ pictured in Figure 11, where $\mathcal{P}(M)$ is the part that will simulate the NB-R-CM M. The locations $\{1_{\mathtt{x}} \mid \mathtt{x} \in X\}$ will allow to encode the values of the different counters during the execution: for a configuration C, $C(1_{\mathtt{x}})$ will represent the value of the counter $\mathtt{x}$. We give then $\mathcal{P}(M) = (Q_M, \Sigma_M, \ell_{in}, \ell_f, T_M)$ with $Q_M = \mathrm{Loc} \cup \{\ell_\delta \mid \delta \in \Delta_b\}$, $\Sigma_M = \{\mathrm{inc}_{\mathtt{x}}, \overline{\mathrm{inc}}_{\mathtt{x}}, \mathrm{dec}_{\mathtt{x}}, \overline{\mathrm{dec}}_{\mathtt{x}}, \mathrm{nbdec}_{\mathtt{x}} \mid \mathtt{x} \in X\}$, and $T_M = \{(\ell_i, !\mathrm{inc}_{\mathtt{x}}, \ell_\delta), (\ell_\delta, ?\overline{\mathrm{inc}}_{\mathtt{x}}, \ell_j) \mid \delta = (\ell_i, \mathtt{x}+, \ell_j) \in \Delta_b\} \cup \{(\ell_i, !\mathrm{dec}_{\mathtt{x}}, \ell_\delta), (\ell_\delta, ?\overline{\mathrm{dec}}_{\mathtt{x}}, \ell_j) \mid \delta = (\ell_i, \mathtt{x}-, \ell_j) \in \Delta_b\} \cup \{(\ell_i, !\mathrm{nbdec}_{\mathtt{x}}, \ell_j) \mid (\ell_i, nb(\mathtt{x}-), \ell_j) \in \Delta_{nb}\} \cup \{(\ell_i, \tau, \ell_j) \mid (\ell_i, \bot, \ell_j) \in \Delta_b\}$. Here, the reception of a message $\overline{\mathrm{inc}}_{\mathtt{x}}$ (respectively $\overline{\mathrm{dec}}_{\mathtt{x}}$) works as an acknowledgement, ensuring that a process has indeed received the message $\mathrm{inc}_{\mathtt{x}}$ (respectively $\mathrm{dec}_{\mathtt{x}}$), and that the corresponding counter has been incremented (resp. decremented). For non-blocking decrement, obviously no acknowledgement is required. The protocol $\mathcal{P} = (Q, \Sigma, q_{in}, \ell_f, T)$ is then defined with $Q = Q_M \cup \{1_{\mathtt{x}}, q_{\mathtt{x}}, q'_{\mathtt{x}} \mid \mathtt{x} \in X\} \cup \{q_{in}, q, q_\bot\}$, $\Sigma = \Sigma_M \cup \{L, R\}$ and T is the set of transitions T_M along with the transitions pictured in Figure 11. Note that there is a transition $(\ell, ?L, q_\bot)$ for all $\ell \in Q_M$.

With two non-blocking transitions on L and R at the beginning, protocol $\mathcal{P}$ can faithfully simulate the NB-R-CM M without further ado, provided that the initial configuration contains enough processes to simulate all the counters values during the execution: after having sent a process in state ℓ_{in}, any transition of M can be simulated in $\mathcal{P}$. Conversely, an initial execution of $\mathcal{P}$ can send multiple processes into the $\mathcal{P}(M)$ zone, which can mess up the simulation. However, each new process entering $\mathcal{P}(M)$ will first send the message

L, and as a consequence the process already in $\{q\} \cup Q_M$, if any, will move to the deadlock state $q_\perp$ receiving this message, and then the new process will send the message R, which will be received by some process in $\{q_{\mathtt{x}}, q'_{\mathtt{x}} \mid \mathtt{x} \in X\}$, if any. Moreover, the construction of the protocol ensures that there can only be one process in the set of states $\{q_{\mathtt{x}}, q'_{\mathtt{x}} \mid \mathtt{x} \in X\}$. Then, if we have reached a configuration simulating the configuration (ℓ, v) of M, sending a new process in the $\mathcal{P}(M)$ zone will lead to a configuration (ℓ_{in}, v), and hence simply mimicks a restore transition of M. So every initial execution of $\mathcal{P}$ corresponds to an initial execution of M.

▶ **Theorem 4.2.** *SCover and CCover over rendez-vous protocols are Expspace complete.*

5 Coverability for Wait-Only Protocols

In this section, we study a restriction on rendez-vous protocols in which we assume that a process waiting to answer a rendez-vous cannot perform another action by itself. This allows for a polynomial time algorithm for solving CCover.

5.1 Wait–Only Protocols

We say that a protocol $\mathcal{P} = (Q, \Sigma, q_{in}, q_f, T)$ is *wait-only* if the set of states Q can be partitioned into Q_A — the *active states* — and Q_W — the *waiting* states — with $q_{in} \in Q_A$ and:

- for all $q \in Q_A$, for all $(q', ?m, q'') \in T$, we have $q' \neq q$;
- for all $q \in Q_W$, for all $(q', !m, q'') \in T$, we have $q' \neq q$ and for all $(q', \tau, q'') \in T$, we have $q' \neq q$.

From a waiting state, a process can only perform receptions (if it can perform anything), whereas in an active state, a process can only perform internal actions or send messages. Examples of wait-only protocols are given by Figures 12 and 13.

In the sequel, we will often refer to the paths of the underlying graph of the protocol. Formally, a *path* in a protocol $\mathcal{P} = (Q, \Sigma, q_{in}, q_f, T)$ is either a control state $q \in Q$ or a finite sequence of transitions in T of the form $(q_0, a_0, q_1)(q_1, a_1, q_2) \dots (q_k, a_k, q_{k+1})$, the first case representing a path from q to q and the second one from q_0 to q_{k+1}.

5.2 Abstract Sets of Configurations

To solve the coverability problem for wait-only protocols in polynomial time, we rely on a sound and complete abstraction of the set of reachable configurations. In the sequel, we consider a wait-only protocol $\mathcal{P} = (Q, \Sigma, q_{in}, q_f, T)$ whose set of states is partitioned into a set of active states Q_A and a set of waiting states Q_W. An *abstract set of configurations* γ is a pair (S, Toks) such that:

- $S \subseteq Q$ is a subset of states, and,
- $\mathit{Toks} \subseteq Q_W \times \Sigma$ is a subset of pairs composed of a waiting state and a message, and,
- $q \notin S$ for all $(q, m) \in \mathit{Toks}$.

We then abstract the set of reachable configurations as a set of states of the underlying protocol. However, as we have seen, some states, like states in Q_A, can host an unbounded number of processes together (this will be the states in S), while some states can only host a bounded number (in fact, 1) of processes together (this will be the states stored in Toks). This happens when a waiting state q answers a rendez-vous m, that has necessarily been requested for a process to be in q. Hence, in Toks, along with a state q, we remember the last message m having been sent in the path leading from q_{in} to q, which is necessarily in Q_W. Observe that, since several paths can lead to q, there can be $(q, m_1), (q, m_2) \in \mathit{Toks}$ with $m_1 \neq m_2$. We denote by Γ the set of abstract sets of configurations.

Let $\gamma = (S, \mathit{Toks})$ be an abstract set of configurations. Before we go into the configurations represented by γ, we need some preliminary definitions. We note $\mathsf{st}(\mathit{Toks})$ the set $\{q \in Q_W \mid$ there exists $m \in \Sigma$ such that $(q, m) \in \mathit{Toks}\}$ of control states appearing in *Toks*. Given a state $q \in Q$, we let $\mathrm{Rec}(q)$ be the set $\{m \in \Sigma \mid$ there exists $q' \in Q$ such that $(q, ?m, q') \in T\}$ of messages that can be received in state q (if q is not a waiting state, this set is empty). Given two different waiting states q_1 and q_2 in $\mathsf{st}(\mathit{Toks})$, we say q_1 and q_2 are *conflict-free* in γ if there exist $m_1, m_2 \in \Sigma$ such that $m_1 \neq m_2$, $(q_1, m_1), (q_2, m_2) \in \mathit{Toks}$ and $m_1 \notin \mathrm{Rec}(q_2)$ and $m_2 \notin \mathrm{Rec}(q_1)$. We now say that a configuration $C \in \mathcal{C}(\mathcal{P})$ *respects* γ if and only if for all $q \in Q$ such that $C(q) > 0$ one of the following two conditions holds:

1. $q \in S$, or,
2. $q \in \mathsf{st}(\mathit{Toks})$ and $C(q) = 1$ and for all $q' \in \mathsf{st}(\mathit{Toks}) \setminus \{q\}$ such that $C(q') = 1$, we have that q and q' are conflict-free.

Note that these conditions only speak about states q such that $C(q) > 0$ as we are only interested in characterising the reachable states (and unreachable states should not appear in S or $\mathsf{st}(\mathit{Toks})$). Let $\llbracket \gamma \rrbracket$ be the set of configurations respecting γ. Note that in $\llbracket \gamma \rrbracket$, for q in S there is no restriction on the number of processes that can be put in q and if q in $\mathsf{st}(\mathit{Toks})$, it can host at most one process. Two states from $\mathsf{st}(\mathit{Toks})$ can both host a process if they are conflict-free.

Finally, we will only consider abstract sets of configurations that are *consistent*. This property aims to ensure that concrete configurations that respect it are indeed reachable from states of S. Formally, we say that an abstract set of configurations $\gamma = (S, \mathit{Toks})$ is *consistent* if (i) for all $(q, m) \in \mathit{Toks}$, there exists a path $(q_0, a_0, q_1)(q_1, a_1, q_2) \dots (q_k, a_k, q)$ in $\mathcal{P}$ such that $q_0 \in S$ and $a_0 = !m$ and for all $1 \leq i \leq k$, we have that $a_i = ?m_i$ and that there exists $(q'_i, !m_i, q''_i) \in T$ with $q'_i \in S$, and (ii) for two tokens $(q, m), (q', m') \in \mathit{Toks}$ either $m \in \mathrm{Rec}(q')$ and $m' \in \mathrm{Rec}(q)$, or, $m \notin \mathrm{Rec}(q')$ and $m' \notin \mathrm{Rec}(q)$. Condition (i) ensures that processes in S can indeed lead to a process in the states from $\mathsf{st}(\mathit{Toks})$. Condition (ii) ensures that if in a configuration C, some states in $\mathsf{st}(\mathit{Toks})$ are pairwise conflict-free, then they can all host a process together.

▶ **Lemma 5.1.** *Given $\gamma \in \Gamma$ and a configuration C, there exists $C' \in \llbracket \gamma \rrbracket$ such that $C' \geq C$ if and only if $C \in \llbracket \gamma \rrbracket$. Checking that $C \in \llbracket \gamma \rrbracket$ can be done in polynomial time.*

5.3 Computing Abstract Sets of Configurations

Our polynomial time algorithm is based on the computation of a polynomial length sequence of consistent abstract sets of configurations leading to a final abstract set characterising in a sound and complete manner (with respect to the coverability problem), an abstraction for the set of reachable configurations. This will be achieved by a function $F : \Gamma \to \Gamma$, that inductively computes this final abstract set starting from $\gamma_0 = (\{q_{in}\}, \emptyset)$. Formal definition of the function F relies on intermediate sets $S'' \subseteq Q$ and $\mathit{Toks}'' \subseteq Q_W \times \Sigma$, which are the smallest sets satisfying the conditions described in Table 1.

From S and *Toks*, rules described in Table 1 add states and tokens to S'' and Toks'' from the outgoing transitions from states in S and $\mathsf{st}(\mathit{Toks})$. It must be that every state added to S'' can host an unbounded number of processes, and every state added to Toks'' can host at least one process, furthermore, two conflict-free states in Toks'' should be able to host at least one process at the same time.

Table 1 Definition of S'', Toks'' for $\gamma = (S, \mathit{Toks})$.

Construction of intermediate states S'' and Toks''

1. $S \subseteq S''$ and $\mathit{Toks} \subseteq \mathit{Toks}''$
2. for all $(p, \tau, p') \in T$ with $p \in S$, we have $p' \in S''$
3. for all $(p, !a, p') \in T$ with $p \in S$, we have:
   - **a.** $p' \in S''$ if $a \notin \mathrm{Rec}(p')$ or if there exists $(q, ?a, q') \in T$ with $q \in S$;
   - **b.** $(p', a) \in \mathit{Toks}''$ otherwise (i.e. when $a \in \mathrm{Rec}(p')$ and for all $(q, ?a, q') \in T$, $q \notin S$);
4. for all $(q, ?a, q') \in T$ with $q \in S$ or $(q, a) \in \mathit{Toks}$, we have $q' \in S''$ if there exists $(p, !a, p') \in T$ with $p \in S$;
5. for all $(q, ?a, q') \in T$ with $(q, m) \in \mathit{Toks}$ with $m \neq a$, if there exists $(p, !a, p') \in T$ with $p \in S$, we have:
   - **a.** $q' \in S''$ if $m \notin \mathrm{Rec}(q')$;
   - **b.** $(q', m) \in \mathit{Toks}''$ if $m \in \mathrm{Rec}(q')$.

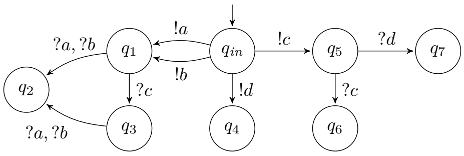


Figure 12 Wait-only protocol $\mathcal{P}_1$.

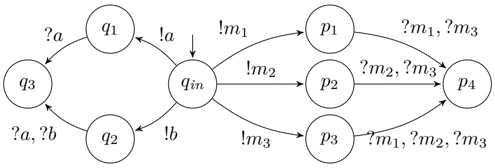


Figure 13 Wait-only protocol $\mathcal{P}_2$.

▶ **Example 5.2.** Consider the wait-only protocol $\mathcal{P}_1$ depicted on Figure 12. From $(\{q_{in}\}, \emptyset)$, rules described in Table 1 construct the following pair $(S''_1, \mathit{Toks}''_1) = (\{q_{in}, q_4\}, \{(q_1, a), (q_1, b), (q_5, c)\})$. In $\mathcal{P}_1$, it is indeed possible to reach a configuration with as many processes as one wishes in the state q_4 by repeating the transition $(q_{in}, !d, q_4)$ (rule 3a). On the other hand, it is possible to put *at most* one process in the waiting state q_1 (rule 3b), because any other attempt from a process in q_{in} will yield a reception of the message a (resp. b) by the process already in q_1. Similarly, we can put at most one process in q_5. Note that in Toks''_1, the states q_1 and q_5 are conflict-free and it is hence possible to have simultaneously one process in both of them.

If we apply rules of Table 1 one more time to $(S''_1, \mathit{Toks}''_1)$, we get $S''_2 = \{q_{in}, q_2, q_4, q_6, q_7\}$ and $\mathit{Toks}''_2 = \{(q_1, a), (q_1, b), (q_3, a), (q_3, b), (q_5, c)\}$. We can put at most one process in q_3: to add one, a process will take the transition $(q_1, ?c, q_3)$. Since (q_1, a), $(q_1, b) \in \mathit{Toks}''_1$, there can be at most one process in state q_1, and this process arrived by a path in which the last request of rendez-vous was $!a$ or $!b$. Since $\{a, b\} \subseteq \mathrm{Rec}(q_3)$, by rule 5b, $(q_3, a), (q_3, b)$ are added. On the other hand we can put as many processes as we want in the state q_7 (rule 5a): from a configuration with one process on state q_5, successive non-blocking request on letter c, and rendez-vous on letter d will allow to increase the number of processes in state q_7.

However, one can observe that q_5 can in fact host an unbounded number of processes: once two processes have been put on states q_1 and q_5 respectively (remember that q_1 and q_5 are conflict-free in $(S''_1, \mathit{Toks}''_1)$), iterating rendez-vous on letter c (with transition $(q_1, ?c, q_3)$) and rendez-vous on letter a put as many processes in state q_5.

As a consequence we need to apply another transformation to $(S''_2, \mathit{Toks}''_2)$ to obtain $F(S''_1, \mathit{Toks}''_1)$. We shall see that this second step has no impact when computing $F((\{q_{in}\}, \emptyset))$ hence we have that $F((\{q_{in}\}, \emptyset)) = (S''_1, \mathit{Toks}''_1)$.

We shall finally get that $F(\gamma)$ is equal to (S', Toks'), where the construction of S' from (S'', Toks''), is given by Table 2 and $\mathit{Toks}' = \mathit{Toks}'' \setminus (S \times \Sigma)$, i.e. all states added to S' are removed from Toks' so a state belongs either to S' or to $\mathsf{st}(\mathit{Toks}')$.

Table 2 Definition of S' where $F(\gamma) = (S', \mathit{Toks}')$ for (S'', Toks'').

Construction of state S', the smallest set including S'' and such that:
6. for all $(q_1, m_1), (q_2, m_2) \in \mathit{Toks}''$ such that $m_1 \neq m_2$ and $m_2 \notin \mathrm{Rec}(q_1)$ and $m_1 \in \mathrm{Rec}(q_2)$, we have $q_1 \in S'$;
7. for all $(q_1, m_1), (q_2, m_2), (q_3, m_2) \in \mathit{Toks}''$ s.t $m_1 \neq m_2$ and $(q_2, ?m_1, q_3) \in T$, we have $q_1 \in S'$;
8. for all $(q_1, m_1), (q_2, m_2), (q_3, m_3) \in \mathit{Toks}''$ such that $m_1 \neq m_2$ and $m_1 \neq m_3$ and $m_2 \neq m_3$ and $m_1 \notin \mathrm{Rec}(q_2)$, $m_1 \in \mathrm{Rec}(q_3)$ and $m_2 \notin \mathrm{Rec}(q_1)$, $m_2 \in \mathrm{Rec}(q_3)$, and $m_3 \in \mathrm{Rec}(q_2)$ and $m_3 \in \mathrm{Rec}(q_1)$, we have $q_1 \in S'$.

▶ **Example 5.3.** Now the case of state q_5 evoked in the previous example leads to application of rule 7, since (q_5, c), $(q_1, a) \in \mathit{Toks}''_2$, and (q_3, a) $(q_1, ?c, q_3) \in T$. Finally, we get that $F(S''_1, \mathit{Toks}''_1) = F(F(\{q_{in}\}, \emptyset)) = (\{q_{in}, q_2, q_4, q_5, q_6, q_7\}, \{(q_1, a), (q_1, b), (q_3, a), (q_3, b)\})$. Since q_1 and q_3 are not conflict-free, they won't be reachable together in a configuration.

We consider now the wait-only protocol $\mathcal{P}_2$ depicted on Figure 13. In that case, to compute $F((\{q_{in}\}, \emptyset))$ we will first have $S'' = \{q_{in}\}$ and $\mathit{Toks}'' = \{(q_1, a), (q_2, b), (p_1, m_1), (p_2, m_2), (p_3, m_3)\}$ (using rule 3b), to finally get $F((\{q_{in}\}, \emptyset)) = (\{q_{in}, q_1, p_1\}, \{(q_2, b), (p_2, m_2), (p_3, m_3)\}))$. Applying rule 6 to tokens (q_1, a) and (q_2, b) from Toks'', we obtain that $q_1 \in S'$: whenever one manages to obtain one process in state q_2, this process can answer the requests on message a instead of processes in state q_1, allowing one to obtain as many processes as desired in state q_1. Now since (p_1, m_1), (p_2, m_2) and (p_3, m_3) are in Toks'' and respect the conditions of rule 8, p_1 is added to the set S' of unbounded states. This case is a generalisation of the previous one, with 3 processes. Once one process has been put on state p_2 from q_{in}, iterating the following actions: rendez-vous over m_3, rendez-vous over m_1, non-blocking request of m_2, will ensure as many processes as one wants on state p_1. Finally applying successively F, we get in this case the abstract set $(\{q_{in}, q_1, q_3, p_1, p_2, p_3, p_4\}, \{(q_2, b)\})$.

We show that F satisfies the following properties.

▶ **Lemma 5.4.**
1. *$F(\gamma)$ is consistent and can be computed in polynomial time for all consistent $\gamma \in \Gamma$.*
2. *If $(S', \mathit{Toks}') = F(S, \mathit{Toks})$ then $S \subseteq S'$ (with $S \neq S'$) or $\mathit{Toks} \subseteq \mathit{Toks}'$.*
3. *For all consistent $\gamma \in \Gamma$, if $C \in [\![\gamma]\!]$ and $C \to C'$ then $C' \in [\![F(\gamma)]\!]$.*
4. *For all consistent $\gamma \in \Gamma$, if $C' \in [\![F(\gamma)]\!]$, then there exists $C'' \in \mathcal{C}$ and $C \in [\![\gamma]\!]$ such that $C'' \geq C'$ and $C \to^* C''$.*

5.4 Polynomial Time Algorithm

We now present our polynomial time algorithm to solve CCover for wait-only protocols. We define the sequence $(\gamma_n)_{n \in \mathbb{N}}$ as follows: $\gamma_0 = (\{q_{in}\}, \emptyset)$ and $\gamma_{i+1} = F(\gamma_i)$ for all $i \in \mathbb{N}$. First note that γ_0 is consistent and that $[\![\gamma_0]\!] = \mathcal{I}$ is the set of initial configurations. Using Lemma 5.4, we deduce that γ_i is consistent for all $i \in \mathbb{N}$. Furthermore, each time we apply F to an abstract set of configurations (S, Toks) either S or Toks increases, or (S, Toks) stabilises. Hence for all $n \geq |Q|^2 * |\Sigma|$, we have $\gamma_{n+1} = F(\gamma_n) = \gamma_n$. Let $\gamma_f = \gamma_{|Q|^2 * |\Sigma|}$. Using Lemma 5.4, we get:

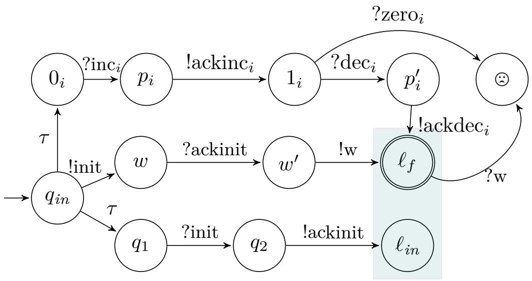


Figure 14 The protocol $\mathcal{P}$ – The coloured zone contains transitions pictured in Figures 15–17.

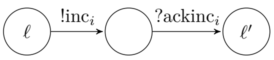


Figure 15 Translation of $(\ell, \mathtt{x}_i{+}, \ell')$.

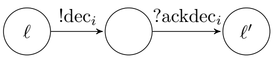


Figure 16 Translation of $(\ell, \mathtt{x}_i{-}, \ell')$.

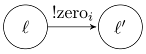


Figure 17 Translation of $(\ell, \mathtt{x}_i{=}0, \ell')$.

▶ **Lemma 5.5.** *Given $C \in \mathcal{C}$, there exists $C_0 \in \mathcal{I}$ and $C' \geq C$ such that $C_0 \rightarrow^* C'$ if and only if there exists $C'' \in [\![\gamma_f]\!]$ such that $C'' \geq C$.*

We need to iterate $|Q|^2 * |\Sigma|$ times the function F to compute γ_f and each computation of F can be done in polynomial time. Furthermore checking whether there exists $C'' \in [\![\gamma_f]\!]$ such that $C'' \geq C$ for a configuration $C \in \mathcal{C}$ can be done in polynomial time by Lemma 5.1, hence using the previous lemma we obtain the desired result.

▶ **Theorem 5.6.** *CCover and SCover restricted to wait-only protocols are in Ptime.*

6 Undecidability of Synchro

It is known that Cover[CM] is undecidable in its full generality [18]. This result holds for a very restricted class of counter machines, namely Minsky machines (Minsky-CM for short), which are CM over 2 counters, $\mathtt{x}_1$ and $\mathtt{x}_2$. Actually, it is already undecidable whether there is an execution $(\ell_{in}, \mathbf{0}_{\{\mathtt{x}_1,\mathtt{x}_2\}}) \rightsquigarrow^* (\ell_f, \mathbf{0}_{\{\mathtt{x}_1,\mathtt{x}_2\}})$. Reduction from this last problem gives the following result.

▶ **Theorem 6.1.** *Synchro is undecidable, even for wait-only protocols.*

Fix $M = (\mathrm{Loc}, \ell_0, \{\mathtt{x}_1, \mathtt{x}_2\}, \Delta)$ with $\ell_f \in \mathrm{Loc}$ the final state. W.l.o.g., we assume that there is no outgoing transition from state ℓ_f in the machine. The protocol $\mathcal{P}$ is described in Figures 14–16. The states $\{0_i, p_i, 1_i, p'_i \mid i = 1, 2\}$ will be visited by processes simulating values of counters, while the states in Loc will be visited by a process simulating the different locations in the Minsky-CM. If at the end of the computation, the counters are equal to 0, it means that each counter has been incremented and decremented the same number of times, so that all processes simulating the counters end up in the state ℓ_f. The first challenge is to appropriately check when a counter equals 0. This is achieved thanks to the non-blocking semantics: the process sends a message !zero$_i$ to check if the counter i equals 0. If it is does not, the message will be received by a process that will end up in the deadlock state ☹. The second challenge is to ensure that only one process simulates the Minsky-CM in the states in Loc. This is ensured by the states $\{w, w'\}$. Each time a process arrives in the ℓ_{in} state, another must arrive in the w' state, as a witness that the simulation has begun. This witness must reach ℓ_f for the computation to be a testifier of a positive instance of Synchro, but it should be the first to do so, otherwise a process already in ℓ_f will receive the message "w" and reach the deadlock state ☹. Thus, if two processes simulate the Minsky-CM, there will be two witnesses, and they won't be able to reach ℓ_f together.

7 Conclusion

We have introduced the model of parameterised networks communicating by non-blocking rendez-vous, and showed that safety analysis of such networks becomes much harder than in the framework of classical rendez-vous. Indeed, CCover and SCover become Expspace-complete and Synchro undecidable in our framework, while these problems are solvable in polynomial time in the framework of [13]. We have introduced a natural restriction of protocols, in which control states are partitioned between *active* states (that allow requesting of rendez-vous) and *waiting* states (that can only answer to rendez-vous) and showed that CCover can then be solved in polynomial time. Future work includes finding further restrictions that would yield decidability of Synchro. A candidate would be protocols in which waiting states can only receive *one* message. Observe that in that case, the reduction of Section 6 can be adapted to simulate a test-free CM, hence Synchro for this subclass of protocols is as hard as reachability in Vector Addition Systems with States, i.e. non-primitive recursive [16]. Decidability remains open though.

References

1 P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS'96*, pages 313–321. IEEE Computer Society, 1996.

2 P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1-2):109–127, 2000.

3 K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

4 A. R. Balasubramanian, J. Esparza, and M. A. Raskin. Finding cut-offs in leaderless rendez-vous protocols is easy. In *FOSSACS'21*, volume 12650 of *LNCS*, pages 42–61. Springer, 2021.

5 G. Delzanno, J. F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded java programs. In *TACAS'02*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

6 G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro. On the complexity of parameterized reachability in reconfigurable broadcast networks. In *FSTTCS'12*, volume 18 of *LIPIcs*, pages 289–300. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.

7 A. Durand-Gasselin, J. Esparza, P. Ganty, and R. Majumdar. Model checking parameterized asynchronous shared-memory systems. *Formal Methods in System Design*, 50(2-3):140–167, 2017.

8 J. Esparza. Decidability and complexity of petri net problems – An introduction. In *Advanced Course on Petri Nets*, pages 374–428. Springer, 1998.

9 J. Esparza. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In Ernst W. Mayr and Natacha Portier, editors, *Proceedings of 31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *LIPIcs*, pages 1–10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014.

10 J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS'99*, pages 352–359. IEEE Comp. Soc. Press, July 1999.

11 J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In *CAV'13*, volume 8044 of *LNCS*, pages 124–140. Springer-Verlag, 2013.

12 A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

13 S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

14 L. Guillou, A. Sangnier, and N. Sznajder. Safety analysis of parameterised networks with non-blocking rendez-vous, 2023. arXiv:2307.04546.

15 F. Horn and A. Sangnier. Deciding the existence of cut-off in parameterized rendez-vous networks. In *CONCUR'20*, volume 171 of *LIPIcs*, pages 46:1–46:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

16 Jérôme Leroux. The reachability problem for petri nets is not primitive recursive. In *FOCS'21*, pages 1241–1252. IEEE, 2021.

17 R.J. Lipton. *The reachability problem requires exponential space*. Research report (Yale University. Department of Computer Science). Department of Computer Science, Yale University, 1976.

18 Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

19 C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.

20 J. F. Raskin and L. Van Begin. Petri nets with non-blocking arcs are difficult to analyze. In *INFINITY'03*, volume 98 of *Electronic Notes in Theoretical Computer Science*, pages 35–55. Elsevier, 2003.

Separability and Non-Determinizability of WSTS

Eren Keskin ✉
TU Braunschweig, Germany

Roland Meyer ✉
TU Braunschweig, Germany

Abstract

There is a recent separability result for the languages of well-structured transition systems (WSTS) that is surprisingly general: disjoint WSTS languages are always separated by a regular language. The result assumes that one of the languages is accepted by a deterministic WSTS, and it is not known whether this assumption is needed. There are two ways to get rid of the assumption, none of which has led to conclusions so far: (i) show that WSTS can be determinized or (ii) generalize the separability result to non-deterministic WSTS languages. Our contribution is to show that (i) does not work but (ii) does. As for (i), we give a non-deterministic WSTS language that we prove cannot be accepted by a deterministic WSTS. The proof relies on a novel characterization of the languages accepted by deterministic WSTS. As for (ii), we show how to find finitely represented inductive invariants without having the tool of ideal decompositions at hand. Instead, we work with closures under converging sequences. Our results hold for upward- and downward-compatible WSTS.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Regular languages

Keywords and phrases WSTS, regular separability, determinization

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.8

Related Version *Full Version*: https://arxiv.org/abs/2305.02736

Funding This work was supported by the DFG project EDS@SYN: Effective Denotational Semantics for Synthesis.

1 Introduction

Czerwiński et al. [16, Theorems 6 and 7] have recently established a separability result for the languages of well-structured transition systems (WSTS) [20, 4, 2, 23] that is surprisingly general. Disjoint WSTS languages are always separated by a regular language: whenever we have $L(U) \cap L(V) = \emptyset$, then there is a regular language R with $L(U) \subseteq R$ and $R \cap L(V) = \emptyset$. The result says that WSTS languages either intersect, or they are far apart in that a finite amount of information is sufficient to distinguish them. Applications abound, we elaborate on this in the related work. Unfortunately, the result comes with a grain of salt: it assumes that one of the WSTS, U or V, is deterministic. All attempts to remove the assumption have failed so far. The assumption is used for a central argument in the proof, namely that inductive invariants can be represented in a finite way. With determinism, these invariants are downward-closed sets in a WQO, and hence decompose into finitely many ideals [31, 21, 22]. This is precisely the finite amount of information needed for regularity.

A strategy to circumvent the assumption would be to show that WSTS can be determinized. Czerwiński et al. already argue in this direction. In [16, Theorem 5], they show that both finitely-branching WSTS and WSTS over so-called ω^2-WQOs can be determinized. Unfortunately, this does not cover all WSTS. To sum up, it is still open whether the regular separability result holds for all WSTS languages, and we do not understand the impact of non-determinism on the expressiveness of the WSTS model.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 8; pp. 8:1–8:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our first contribution is to prove the regular separability result for all WSTS languages, without the assumption of determinism. We accept the fact that determinizing a WSTS no longer yields a WSTS, and carefully study the resulting class of transition systems. They are formed over a lattice in which sequences have subsequences that converge in a natural sense. This leads us to define the closure of a set by adding the limits of all converging sequences. The key insight is that the closure of an inductive invariant is again an inductive invariant. Together with the fact that closed sets have finitely many maximal elements, we arrive at the desired finite representation. In short, when moving from WQOs to converging lattices, maximal elements of closed sets form an alternative to ideal decompositions of downward-closed sets. We call the new transition systems converging.

Our second contribution is to show that WSTS cannot be determinized in general. We give a WSTS language T that we prove cannot be accepted by a deterministic WSTS. The proof relies on a novel characterization of the deterministic WSTS languages: they are precisely the languages whose Nerode (right) quasi order is a WQO. The characterization provides a first hint on how to construct T. The language should have an infinite antichain in the Nerode quasi order, for then this cannot be a WQO. The second hint stems from the determinizability result [16, Theorem 5]. The accepting WSTS should be infinitely branching and the WQO should be no ω^2-WQO. Such WQOs embed the so-called Rado WQO [8, Section 2]. Moreover, the Rado WQO is known to have an infinite antichain when constructing downward-closed sets [22, Proposition 4.2]. The definition of T is thus guided by the idea of translating the Rado antichain into an antichain in the Nerode quasi order. Interestingly, the underlying WSTS is deterministic except for the choice of the initial state.

We develop these results for upward-compatible WSTS [23]. Our third contribution is to show that they also hold for downward-compatible WSTS. We achieve this by proving general relationships between the models. A key insight is that the complement of a deterministic upward-compatible WSTS is a deterministic downward-compatible WSTS. Moreover, the reversal of an upward-compatible WSTS language is a downward-compatible WSTS language.

Details and proofs missing here can be found in the full version of this article [32].

Related Work. The converging transition systems (CTS) we use to generalize the regular separability result [16] have a topological flavor, and indeed are inspired by Goubault-Larrecq's Noetherian transition systems [26, 27]. One difference is that we had to formulate CTS in lattice-theoretic terms to be able to import a theorem from [16] that links regular separability to the existence of finitely represented inductive invariants. Another difference is the study of such invariants (we prove stability under closure) that has no analogue in [26, 27].

We show that deterministic WSTS accept a strictly weaker class of languages than their non-deterministic counterparts. The work [3] also compares classes of WSTS languages, but for fixed models (extended Petri nets). We allow the determinization to freely select the WQO and the transitions, meaning we have considerably less syntactic constraints to work with. There are also pumping lemmas to distinguish WSTS languages from (among others) context-free languages [24]. Our characterization of the deterministic WSTS languages is stronger than the necessary conditions in pumping lemmas. Our language witnessing the weakness of deterministic WSTS is accepted by an infinitely-branching WSTS, a class of systems studied in [7]. That work concentrates on decidability results and pays attention to effectiveness, while we prove a statement of existence and do not need such assumptions.

There is recent interest in separability problems for infinite-state systems [17, 40, 14, 39, 12]. One reason is that standard algorithms rarely apply to separability problems, but these problems tend to call for new approaches. With the basic separator technique [18], Czerwiński and Zetzsche have shown that there is hope for general methods that apply to a range of separability problems [10, 11, 15]. With the closure of inductive invariants under converging sequences, we hope to also have contributed a versatile tool.

Another reason for the popularity of separability problems is their usefulness in verification. In [1], separators act as interpolants in abstraction-guided verification [9]. In [6], separators are advocated as interfaces in rely-guarantee reasoning [30]. In this context, our result implies that regular interfaces yield a complete proof method, provided the system is well-structured.

2 Well-Structured Transition Systems

We recall well-structured transition systems (WSTS) with upward compatibility [20, 4, 2, 23]. Downward compatibility will be addressed in Section 5.

Orders. Let $(Q, \leq)$ be a quasi order and $P \subseteq Q$. We call P a chain, if $\leq$ restricted to P is a total order. We call P an antichain, if the elements in P are pairwise incomparable. The upward closure of P is $\uparrow P = \{q \in Q \mid \exists p \in P.\ p \leq q\}$. We call P upward closed, if $P = \uparrow P$. The powerset of Q restricted to the upward-closed sets is $\mathbb{U}(Q)$. The downward closure is defined similarly and we use $\mathbb{D}(Q)$ for the downward-closed sets. We call $(Q, \leq)$ a well quasi order (WQO), if for every infinite sequence $[p_i]_{i \in \mathbb{N}}$ in Q there are indices $i < j$ with $p_i \leq p_j$.

Let $(Q, \leq)$ be a partially-ordered set. We write $\max P$ for the set of maximal elements in the subset $P \subseteq Q$. They may not exist, in which case the set is empty. We call $(Q, \leq)$ a complete lattice, if all $P \subseteq Q$ have a greatest lower bound in Q, also called meet and denoted by $\bigsqcap P \in Q$, and a least upper bound in Q, also called join and denoted by $\bigsqcup P \in Q$. A function $f : Q \to Q$ on a complete lattice is join preserving [13, Section 11.4], if it distributes over arbitrary joins in that $f(\bigsqcup P) = \bigsqcup f(P)$ for all $P \subseteq Q$, where $f(P) = \{f(p) \mid p \in P\}$. We call $(Q, \leq)$ a completely distributive lattice, if it is a complete lattice where arbitrary meets distribute over arbitrary joins, and vice versa:

$$\bigsqcap_{a \in A} \bigsqcup_{b \in B_a} p_{a,b} = \bigsqcup_{f \in C_{A,B}} \bigsqcap_{a \in A} p_{a,f(a)} \qquad \bigsqcup_{a \in A} \bigsqcap_{b \in B_a} p_{a,b} = \bigsqcap_{f \in C_{A,B}} \bigsqcup_{a \in A} p_{a,f(a)} \ .$$

The definition makes use of the Axiom of Choice: $C_{A,B}$ denotes the set of choice functions that map each $a \in A$ to a choice $b \in B_a$. It is also important to note that, for any quasi order $(Q, \leq)$, $(\mathbb{D}(Q), \subseteq)$ is a completely distributive lattice.

Labeled Transition Systems. A labeled transition system (LTS) is a tuple $U = (Q, I, \Sigma, \delta, F)$ that consists of a set of states Q, in our setting typically infinite, a set of inital states $I \subseteq Q$, a set of final states $F \subseteq Q$, a finite alphabet Σ, and a set of labeled transitions $\delta : Q \times \Sigma \to \mathbb{P}(Q)$. The LTS is deterministic, if $|I| = |\delta(p, a)| = 1$ for all $p \in Q$ and $a \in \Sigma$.

Its language is the set of words that can reach a final state from an initial state:

$$L(U) = \{w \in \Sigma^* \mid \delta(I, w) \cap F \neq \emptyset\} \ .$$

Here, we extend the transition relation to sets of states and words: $\delta(P, w.a) = \delta(\delta(P, w), a)$ and $\delta(P, a) = \bigcup_{p \in P} \delta(p, a)$. Finally, if the LTS is deterministic, we write $(Q, y, \Sigma, \delta, F)$ and $\delta(p, a) = q$ instead of $(Q, \{y\}, \Sigma, \delta, F)$ and $\delta(p, a) = \{q\}$.

Let U_1 and U_2 be LTS with $U_i = (Q_i, I_i, \Sigma, \delta_i, F_i)$. We define their synchronized product to be the LTS $U_1 \times U_2 = (Q_1 \times Q_2, I_1 \times I_2, \Sigma, \delta, F_1 \times F_2)$ where $(q_1, q_2) \in \delta((p_1, p_2), a)$, if $q_1 \in \delta_1(p_1, a)$ and $q_2 \in \delta_2(p_2, a)$. Then $L(U_1 \times U_2) = L(U_1) \cap L(U_2)$.

Compatibility. We work with LTS $U = (Q, I, \Sigma, \delta, F)$ whose states form a quasi order $(Q, \leq)$ that is compatible with the remaining components as follows. We have $F = \uparrow F$, the final states are upward closed wrt. $\leq$. Moreover, $\leq$ is a simulation relation [36]: for all pairs of related states $p_1 \leq q_1$ and for all letters $a \in \Sigma$ we have:

for all $p_2 \in \delta(p_1, a)$ there is $q_2 \in \delta(q_1, a)$ with $p_2 \leq q_2$.

We also make the quasi order explicit and call $U = (Q, \leq, I, \Sigma, \delta, F)$ an *upward-compatible* LTS (ULTS).

ULTS can be determinized, in the case of U this yields

$$U^{det} \quad = \quad (\mathbb{D}(Q), \subseteq, \downarrow I, \Sigma, \delta^{det}, F^{det}) \ .$$

The states are the downward-closed sets ordered by inclusion, the transition relation is defined by closing the result of the original transition relation downwards, $\delta^{det}(D, a) = \downarrow\delta(D, a)$ for all $D \in \mathbb{D}(Q)$ and $a \in \Sigma$, and the set of final states consists of all downward-closed sets that contain a final state in the original ULTS, $F^{det} = \{D \in \mathbb{D}(Q) \mid D \cap F \neq \emptyset\}$.

▶ **Lemma 1.** *Let U be an ULTS. Then U^{det} is a deterministic ULTS with $L(U^{det}) = L(U)$.*

We write detULTS for the class of deterministic ULTS. The synchronized product of ULTS is again an ULTS (with the product order).

Well-Structuredness. An *upward-compatible well-structured transition system* (WSTS) is an ULTS U whose states $(Q, \leq)$ form a WQO. The synchronized product of WSTS is again a WSTS. We are interested in $L(\text{WSTS})$, the class of all languages accepted by WSTS. We also study $L(\text{detWSTS}) \subseteq L(\text{WSTS})$, the class of languages accepted by deterministic WSTS.

We observe that we can focus on WSTS with a countable number of states.

▶ **Lemma 2.** *For every $L \in L(WSTS)$ there is a WSTS U with a countable number of states so that $L = L(U)$.*

The lemma needs two arguments: the language consists of a countable number of words, and we can assume the transition relation to yield downward-closed sets.

3 Regular Separability of WSTS Languages

Two languages $L_1, L_2 \subseteq \Sigma^*$ are *separable by a regular language*, denoted by $L_1 \mid L_2$, if there is a regular language $R \subseteq \Sigma^*$ with $L_1 \subseteq R$ and $R \cap L_2 = \emptyset$. Our main result is that disjoint WSTS languages are always separable in this sense.

▶ **Theorem 3.** *For $L_1, L_2 \in L(WSTS)$, we have $L_1 \mid L_2$ if and only if $L_1 \cap L_2 = \emptyset$.*

The conclusion is the same as in the main theorem of [16], but we do not need the premise that one of the languages is accepted by a deterministic WSTS. The implication from left to right is trivial, the implication from right to left is our first contribution.

We summarize the arguments. The plan is to invoke the proof principle for regular separability in [16, Theorem 11] and show that the product system has a finitely represented inductive invariant. The principle holds for general ULTS but needs one of them deterministic. Therefore, our first step is to determinize the given WSTS. Determinizing a WSTS will yield an ULTS, but may ruin the WQO property. We show that the set of states of the resulting ULTS still has a rich structure: it is a powerset lattice in which every infinite sequence contains a subsequence that converges in a natural sense. We call such ULTS converging transition systems (CTS). We only define CTS as deterministic models, which is why we determinize both WSTS. CTS are closed under products. Moreover, since the initial languages are disjoint by the assumption, the product trivially has an inductive invariant. It thus remains to turn this invariant of the product into an invariant that can be represented in a finite way. The idea is to add the limits of all converging sequences in the invariant. Since the CTS transitions are compatible with limits, the resulting set of states is again an inductive invariant. By Zorn's lemma, every set can contain only finitely many maximal elements. The maximal elements thus form the finite representation that was needed to conclude the proof.

It would be possible to give the proof at a set-theoretic level, by explicitly working with products of powerset lattices. CTS allow us to abstract away the product structure and highlight the key arguments in the limit construction. We turn to the details.

3.1 Proof Principle for Regular Separability

To establish regular separability, we rely on a proof principle introduced in [16]. The notion of an inductive invariant will be recalled in a moment.

▶ **Theorem 4** (Proof principle for regular separability, [16, Theorem 11]). *Consider ULTS U, V, one deterministic. If $U \times V$ has a finitely represented inductive invariant, then $L(U) \mid L(V)$.*

Interestingly, the proof principle does not need the WQO assumption of WSTS but holds for general ULTS. It does assume one of the ULTS to be deterministic, though. Recall that an *inductive invariant* for an ULTS $(Q, \leq, I, \Sigma, \delta, F)$ is a downward-closed set of states $S \subseteq Q$ that includes all initial states, excludes all final states, and is closed under taking transitions:

$$I \subseteq S \qquad S \cap F = \emptyset \qquad \delta(S, a) \subseteq S\ .$$

The inductive invariant is *finitely represented*, if there is a finite set $C \subseteq S$ with $S = {\downarrow}C$. We refer to a set C that satisfies this as a *cover* of S.

When trying to invoke Theorem 4, finding an inductive invariant for $U \times V$ is easy: the invariant is guaranteed to exist as soon as the language $L(U \times V) = L(U) \cap L(V)$ is empty, which is precisely the hypothesis we start from.

▶ **Lemma 5** ([16, Lemma 10]). *An ULTS U admits an inductive invariant iff $L(U) = \emptyset$.*

The difficult part is to find an inductive invariant that can be represented in a finite way. In [16], this was addressed with ideal decompositions [31, 21, 22]. The ideal decompositions, however, needed the WQO assumption, which lead to the requirement in the main theorem that one WSTS had to be deterministic. As we show in Section 4, this is a real restriction: there are WSTS languages that cannot be accepted by a deterministic WSTS.

Our contribution is to find finitely represented inductive invariants without making use of ideal decompositions. Our approach is to determinize the given WSTS with the construction in Lemma 1, and accept that we can no longer guarantee the result to be a WSTS.

3.2 Converging Transition Systems: WSTS in Disguise

We propose converging transition systems (CTS), a new class of ULTS that is general enough to capture determinized WSTS and retains enough structure to establish the existence of finitely represented inductive invariants. CTS are inspired by Noetherian transition systems [26, 27], but are formulated in a lattice-theoretic rather than in a topological way.

Recall that determinized WSTS have as state space $(\mathbb{D}(Q), \subseteq)$, where $(Q, \leq)$ is a WQO. In a WQO, every infinite sequence admits an increasing subsequence. It is well known [38] that this may not hold for $(\mathbb{D}(Q), \subseteq)$. However, a natural relaxation holds: every infinite sequence $[X_i]_{i\in\mathbb{N}}$ admits an infinite subsequence $[X_{\varphi(i)}]_{i\in\mathbb{N}}$, where any element that is present in one set is present in almost every set. A similar property, defined for complete lattices, is called convergence in the literature [25]. Our definition differs from the citation in two ways. We restrict ourselves to sequences (as opposed to nets), and we require convergence to the join (as opposed to $\limsup = \liminf$). This suffices for our setting.

▶ **Definition 6.** *A* converging lattice $(Q, \leq)$ *is a completely distributive lattice, where every sequence* $[p_i]_{i\in\mathbb{N}}$ *has a converging subsequence* $[p_{\varphi(i)}]_{i\in\mathbb{N}}$*. A* converging sequence $[q_i]_{i\in\mathbb{N}}$ *is an infinite sequence with*

$$\bigsqcup_{i\in\mathbb{N}} \bigsqcap_{j\geq i} q_j = \bigsqcup_{i\in\mathbb{N}} q_i \ .$$

The equality formalizes our explaination from before. In the context of sets, where join and meet are respectively union and intersection, the right-hand side of the equation contains all elements that appear in any set in the sequence. The left side iterates over every finite initial segment, and includes every element that appears in all sets outside of this segment. Every element that is missing in only finitely many sets will eventually be included.

Converging lattices not only generalize downward-closed subsets of WQOs, they are also a sufficient condition for them. The backward direction is by [38, Proof of Theorem 3]. The forward direction is by an application of the following fact [38], also [33, Fact III.3]: $(\mathbb{D}(Q), \subseteq)$ is well-founded if and only if the order is a WQO. The details are given in [32].

▶ **Lemma 7.** $(\mathbb{D}(Q), \subseteq)$ *is a converging lattice if and only if* $(Q, \leq)$ *is a WQO.*

The space of converging sequences is closed under the application of join preserving functions as formulated next. While we would expect this result to be known, we have not found a reference. The lemma is central to our argument, therefore we give the proof.

▶ **Lemma 8.** *Let* $(Q, \leq)$ *be a lattice,* $[p_i]_{i\in\mathbb{N}}$ *a converging sequence in* Q*, and* $f : Q \to Q$ *a join preserving function. Then also* $[f(p_i)]_{i\in\mathbb{N}}$ *is converging.*

Proof. Due to convergence of the given sequence, we have $\bigsqcup_{i\in\mathbb{N}} \bigsqcap_{j\geq i} p_j = \bigsqcup_{i\in\mathbb{N}} p_i$. This equality yields $f(\bigsqcup_{i\in\mathbb{N}} \bigsqcap_{j\geq i} p_j) = f(\bigsqcup_{i\in\mathbb{N}} p_i)$. By join preservation of f, we get

$$\bigsqcup_{i\in\mathbb{N}} f(\bigsqcap_{j\geq i} p_j) = \bigsqcup_{i\in\mathbb{N}} f(p_i) \ .$$

Function f is not assumed to be meet preserving. But we can show an inequality that is sufficient for our needs. For all $S \subseteq Q$ and $s \in S$, we have $f(\bigsqcap S) \leq f(s) \sqcup f(\bigsqcap S)$. Join preservation and the fact that $s \in S$ yield $f(s) \sqcup f(\bigsqcap S) = f(s \sqcup \bigsqcap S) = f(s)$. We have thus shown $f(\bigsqcap S) \leq f(s)$ for all $s \in S$. This means $f(\bigsqcap S) \leq \bigsqcap_{s\in S} f(s)$.

We apply this inequality to the previous equality:

$$\bigsqcup_{i\in\mathbb{N}} f(p_i) = \bigsqcup_{i\in\mathbb{N}} f(\bigsqcap_{j\geq i} p_j) \leq \bigsqcup_{i\in\mathbb{N}} \bigsqcap_{j\geq i} f(p_j) \leq \bigsqcup_{i\in\mathbb{N}} f(p_i) \ .$$

This is $\bigsqcup_{i\in\mathbb{N}} \bigsqcap_{j\geq i} f(p_j) = \bigsqcup_{i\in\mathbb{N}} f(p_i)$, as desired. ◀

We explain the considerations that lead us to the definition of CTS given below. In the light of Lemma 7, the states of a CTS should form a converging lattice. This, however, was not enough to guarantee the existence of finitely represented inductive invariants. One requirement of invariants is that they are closed under taking transitions. To understand which sets satisfy this, we had to restrict the transition relation. We define CTS only as a deterministic model. Then the transitions form a function $\delta(-, a)$ for every letter $a \in \Sigma$. Upward compatibility of these functions is not very informative. Consider determinized WSTS: upward compatibility gives us $\delta(S_0 \cup S_1, a) \supseteq \delta(S_0, a)$, while we expect $\delta(S_0 \cup S_1, a) = \delta(S_0, a) \cup \delta(S_1, a)$. In lattice-theoretic terms, we expect the transition functions $\delta(-, a)$ to be join preserving. A benefit of this requirement is of course that it makes Lemma 8 available. An invariant should also be disjoint from the final states so that we had to control this set as well. When determinizing WSTS, a set $D \in \mathbb{D}(Q)$ is final as soon as it contains a single final state. Given the definition of convergence, we relax this to containing a finite set of final states.

▶ **Definition 9.** *A* converging transition system *(CTS) is an ULTS $U = (Q, \leq, y, \Sigma, \delta, F)$ that is deterministic, where $(Q, \leq)$ is a converging lattice, the functions $\delta(-, a)$ are join preserving for all $a \in \Sigma$, and the final states satisfy*

finite acceptance: for every $\bigsqcup K \in F$ there is a finite set $N \subseteq K$ with $\bigsqcup N \in F$.

The determinization of a WSTS yields a CTS, as it was one of the goals of the CTS definition. Somewhat surprisingly, CTS do not add expressiveness but their languages are already accepted by (non-deterministic) WSTS. The construction is via join prime elements and can be found in the full version [32]. Together, the CTS languages are precisely the WSTS languages, and one may see Definition 9 as a reformulation of the WSTS model.

▶ **Proposition 10.** *If U is a WSTS, then U^{det} is a CTS. For every CTS V, there is a WSTS U with $L(V) = L(U)$. Together, $L(WSTS) = L(CTS)$.*

The correspondence allows us to import the countability assumption from Lemma 2. Indeed, if the WQO $(Q, \leq)$ is countable, then there is only a countable number of downward-closed sets in $(\mathbb{D}(Q), \subseteq)$. This is by a standard argument for WSTS: each downward-closed set can be characterized by its complement, the complement is upward closed, and is therefore characterized by its finite set of minimal elements.

▶ **Lemma 11.** *For every $L \in L(CTS)$, there is a CTS U over a countable number of states so that $L = L(U)$*

We will also need that CTS are closed under synchronized products.

▶ **Lemma 12.** *If U and V are CTS, so is $U \times V$.*

We summarize the findings so far. Given disjoint WSTS languages $L(V_1) \cap L(V_2) = \emptyset$, the goal is to show regular separability $L(V_1) \mid L(V_2)$. We first determinize both WSTS. By Proposition 10, V_1^{det} and V_2^{det} are CTS. Moreover, by Lemma 1, determinization preserves the language. We use Lemma 11 to obtain countable CTS U_1 and U_2 that accept the same languages. To show regular separability, we now intend to invoke Theorem 4 on U_1 and U_2. CTS are already deterministic. It thus remains to show that $U_1 \times U_2$ has a finitely represented inductive invariant. With Lemma 12, $U_1 \times U_2$ is another CTS U. Moreover, the product corresponds to language intersection, so $L(U) = \emptyset$. By Lemma 5, we know that U has an inductive invariant. We now show how to turn this invariant into a finitely represented one.

3.3 Inductive Invariants in CTS

We show the following surprising property for countable CTS: every inductive invariant S can be generalized to an inductive invariant $cl(S)$ that is finitely represented. The closure operator is defined by adding to S the joins of all converging sequences:

$$cl(S) \quad = \quad \{ \bigsqcup_{i\in\mathbb{N}} p_i \mid [p_i]_{i\in\mathbb{N}} \text{ a converging sequence in } S \} \ .$$

▶ **Proposition 13.** *Let U be a countable CTS and S an inductive invariant of U. Then also $cl(S)$ is an inductive invariant of U and it is finitely represented.*

The proposition concludes the proof of Theorem 3. We simply invoke it on the inductive invariant that exists by Lemma 5 as discussed above. The rest of the section is devoted to the proof. We fix a countable CTS $U = (Q, \leq, y, \Sigma, \delta, F)$ and an inductive invariant $S \subseteq Q$.

As Lemma 14 states, the closure is expansive and idempotent. This means further applications do not add new limits. Here, we need the fact that we have a completely distributive lattice. Moreover, the closure yields a downward-closed set. The closure is also trivially monotonic, and hence an upper closure operator indeed [13, Section 11.7], but we will not need monotonicity. The proof of Lemma 14 is given in the full version [32].

▶ **Lemma 14.** $S \subseteq cl(S) = cl(cl(S)) = {\downarrow}cl(S)$.

Towards showing Proposition 13, we first argue for invariance.

▶ **Lemma 15.** *$cl(S)$ is an inductive invariant.*

Proof. To prove that $cl(S)$ is an inductive invariant, we must show two properties for the joins $\bigsqcup_{i\in\mathbb{N}} p_i = p$ of converging sequences $[p_i]_{i\in\mathbb{N}}$ in S that we added. First, we must show that we do not leave $cl(S)$ when taking transitions, $\delta(p, a) \in cl(S)$ for all $a \in \Sigma$. Second, we must show that the join is not a final state. We begin with the latter. Towards a contradiction, suppose $p \in F$. Convergence yields $\bigsqcup_{i\in\mathbb{N}} \bigsqcap_{j\geq i} p_j \in F$. By the finite acceptance property of CTS, there must be a finite set $K \subseteq \mathbb{N}$ with $k = \max K$ so that

$$\bigsqcup_{i\in K} \bigsqcap_{j\geq i} p_j = \bigsqcap_{j\geq k} p_j \in F \ .$$

Since $\bigsqcap_{j\geq k} p_j \leq p_k$ and F is upward closed, we obtain $p_k \in F$. This is a contradiction: p_k belongs to the inductive invariant S and the invariant does not intersect the final states.

To show $\delta(p, a) \in cl(S)$, we first note that $\delta(p_i, a) \in S$ for all $i \in \mathbb{N}$. This holds as S is an invariant and $p_i \in S$. We now argue that not only the sequence $[\delta(p_i, a)]_{i\in\mathbb{N}}$ is in S, but also its join is in the closure. We use that the transition function $\delta(-, a)$ is join preserving. This allows us to apply Lemma 8 showing that $[\delta(p_i, a)]_{i\in\mathbb{N}}$ coverges. Since the sequence belongs to S, we obtain $\bigsqcup_{i\in\mathbb{N}} \delta(p_i, a) \in cl(S)$. We conclude by applying join preservation:

$$\delta(p, a) = \delta(\bigsqcup_{i\in\mathbb{N}} p_i, a) = \bigsqcup_{i\in\mathbb{N}} \delta(p_i, a) \in cl(S) \ . \qquad ◀$$

It only remains to show that $cl(S)$ is finitely represented.

▶ **Proposition 16.** *There is a finite set $C \subseteq cl(S)$ so that ${\downarrow}C = cl(S)$.*

We break down the proof of Proposition 16 into two steps. First, we show that $cl(S)$ can be covered by an antichain. Then, we show that infinite antichain covers do not exist. This implies that there must be a finite antichain cover. The proofs reasons over *closed* sets, sets that contain the limits of their converging sequences. We rely on the fact that closed sets have at least one maximal element.

▶ **Lemma 17.** *Consider $G \subseteq Q$ closed and non-empty. Then $\max G \neq \emptyset$.*

Moreover, closedness remains intact after certain removals.

▶ **Lemma 18.** *Consider $G, H \subseteq Q$ where G is closed. Then $G \setminus {\downarrow}H$ is closed.*

We postpone the proofs of these lemmas until after the proof of Proposition 16.

▶ **Lemma 19.** *There is an antichain cover of $cl(S)$.*

Proof. We claim that the maximal elements $\max cl(S)$ form an antichain cover of $cl(S)$. It is clear that $\max cl(S)$ is an antichain. Since $cl(S)$ is downward closed by Lemma 14, we also have ${\downarrow}(\max cl(S)) \subseteq cl(S)$. To see that $\max cl(S)$ is a cover, let $G = cl(S) \setminus {\downarrow}(\max cl(S))$ and suppose $G \neq \emptyset$. Lemma 18 tells us that G is closed. By Lemma 17, we get $\max G \neq \emptyset$. Consider $p \in \max G$. By the definition of G, we have $p \notin \max cl(S)$. Then, however, there must be $q \in cl(S)$ with $p \leq q$ and $p \neq q$. If $q \in {\downarrow}(\max cl(S))$, then $p \in {\downarrow}(\max cl(S))$ as well, which is a contradiction to $p \in G$. If conversely $q \in cl(S) \setminus {\downarrow}(\max cl(S)) = G$, then we have a contradiction to $p \in \max G$. ◀

Now we prove the second part of Proposition 16, which states that there can be no infinite antichain cover.

▶ **Lemma 20.** *There is no infinite antichain cover of $cl(S)$.*

Proof. Suppose there is an infinite antichain cover $C \subseteq cl(S)$. Then, there is an infinite sequence $[p_i]_{i\in\mathbb{N}}$ in C. By Definition 6, it has an infinite converging subsequence $[p_{\varphi(i)}]_{i\in\mathbb{N}}$. The closure operator adds $\bigsqcup_{i\in\mathbb{N}} p_{\varphi(i)}$ to $cl(S)$. Since C is a cover of $cl(S)$, there must be $q \in C$ with $\bigsqcup_{i\in\mathbb{N}} p_{\varphi(i)} \leq q$. Because $p_{\varphi(i)} \sqcup p_{\varphi(0)} \leq q$ and $p_{\varphi(i)}$, $p_{\varphi(0)}$ are incomparable, we have $p_{\varphi(i)} < q$ for all $i \in \mathbb{N}$. So $p_{\varphi(i)} < q$ for all $i \in \mathbb{N}$, while at the same time $q, p_{\varphi(i)} \in C$. This contradicts the antichain property. ◀

We conclude by showing Lemma 17 and 18.

Proof of Lemma 17. Let $\emptyset \neq G \subseteq Q$ be closed. We prove G chain complete, meaning for every chain $P \subseteq G$ the limit $\bigsqcup P$ is again in G. Then Zorn's lemma [29] applies and yields $\max G \neq \emptyset$. We have Zorn's lemma, because we agreed on the Axiom of Choice. Towards chain completeness, consider an increasing sequence $[p_i]_{i\in\mathbb{N}}$ in G. We prove that $\bigsqcup_{i\in\mathbb{N}} p_i \in G$. For any $i \in \mathbb{N}$, we have $\bigsqcap_{j\geq i} p_i = p_i$. Hence, replacing each meet with the smallest element shows convergence. Since $[p_i]_{i\in\mathbb{N}}$ converges and G is closed, we have $\bigsqcup_{i\in\mathbb{N}} p_i \in G$.

Although we are in a countable setting, the argument for sequences does not yet cover all chains. The problem is that the counting processs may not respect the order. To see this, consider a chain $P \subseteq G$ of ordinal size $|P| = \omega \cdot 2$. The chain is countable, but no counting process can respect the order. We now argue that still $\bigsqcup P \in G$. By [34, Theorem 1], there is a (wrt. inclusion) increasing sequence of subsets $[P_i]_{i\in\mathbb{N}}$ in $\mathbb{P}(P)$, where each P_i is finite and $\bigcup_{i\in\mathbb{N}} P_i = P$. Finite chains contain maximal elements, so let $p_i = \max P_i = \bigsqcup P_i$. Then

$$\bigsqcup P \;=\; \bigsqcup \bigcup_{i\in\mathbb{N}} P_i \;=\; \bigsqcup_{i\in\mathbb{N}} \bigsqcup P_i \;=\; \bigsqcup_{i\in\mathbb{N}} p_i \,.$$

Since $[P_i]_{i\in\mathbb{N}}$ is an increasing sequence, $[p_i]_{i\in\mathbb{N}}$ is also an increasing sequence. As we have shown before, $\bigsqcup_{i\in\mathbb{N}} p_i \in G$. This concludes the proof. ◀

Proof of Lemma 18. Consider $G, H \subseteq Q$ with G closed. We show that $G \setminus \downarrow H$ is closed. Let $[p_i]_{i\in\mathbb{N}}$ be a converging sequence in $G \setminus \downarrow H$. Let $q = \bigsqcup_{i\in\mathbb{N}} p_i$ and suppose $q \notin G \setminus \downarrow H$. Since G is closed, $q \in G$. Then necessarily $q \in \downarrow H$. But by definition, $p_i \leq q$ for all $i \in \mathbb{N}$. So $p_i \in \downarrow H$ as well. This contradicts the fact that the sequence $[p_i]_{i\in\mathbb{N}}$ lives in $G \setminus \downarrow H$. ◀

4 Non-Determinizability of WSTS

We show that the detWSTS languages form a strict subclass of the WSTS languages. To this end, we define a WSTS language T that we prove cannot be accepted by a detWSTS. The proof relies on a novel characterization of the detWSTS languages that may be of independent interest. In the following, we call T the *witness language*. This is our second main result.

▶ **Theorem 21.** *$\mathcal{L}(detWSTS) \neq \mathcal{L}(WSTS)$.*

Towards the definition of T, recall that finitely-branching WSTS and WSTS over so-called ω^2-WQOs can be determinized [16, Theorem 5]. Moreover, it is known that ω^2-WQOs are precisely the WQOs that do not embed the Rado WQO [8, Section 2]. This suggests we should accept the witness language T by an infinitely-branching WSTS over the Rado WQO. We begin with our characterization of the detWSTS languages, as it will provide additional guidance in the definition of the witness language.

4.1 Characterization of the detWSTS Languages

Our characterization is based on a classical concept in formal languages [28, Theorem 3.9]. The *Nerode quasi order* $\leq_L \subseteq \Sigma^* \times \Sigma^*$ of a language $L \subseteq \Sigma^*$ is defined by $w \leq_L v$, if

for all $u \in \Sigma^*$ we have that $w.u \in L$ implies $v.u \in L$.

The characterization says that the detWSTS languages are precisely the languages whose Nerode quasi order is a WQO. Note that this is not the folklore result [5, Proposition 5.1] saying that a language is regular if and only if the syntactic quasi order is a WQO.

▶ **Lemma 22** (Characterization of $\mathcal{L}$(detWSTS)). *$L \in \mathcal{L}(detWSTS)$ iff $\leq_L$ is a WQO.*

Proof. ⇒ Let $L = L(U)$ with $U = (Q, \leq, i, \Sigma, \delta, F)$ a detWSTS. We extend the order $\leq\, \subseteq Q \times Q$ on the states to an order $\leq_U\, \subseteq \Sigma^* \times \Sigma^*$ on words by setting $w \leq_U v$, if $\delta(i, w) = p$ and $\delta(i, v) = q$ with $p \leq q$. Since U is deterministic, p and q are guaranteed to exist and be unique. It is easy to see that $\leq_U$ is a WQO. We now show that $\leq_U$ is included in the Nerode quasi order, and so also $\leq_L$ is a WQO. To this end, we consider $w \leq_U v$ and $u \in \Sigma^*$ with $w.u \in L$, and show that also $v.u \in L$. We have $\delta(i, w.u) = \delta(p_1, u) = p_2$ and $\delta(i, v.u) = \delta(q_1, u) = q_2$ with $p_1 = \delta(i, w)$ and $q_1 = \delta(i, v)$. Since $w \leq_U v$, we have $p_1 \leq q_1$. With the simulation property of WSTS, this implies $p_2 \leq q_2$. Since $w.u \in L$ and $L = L(U)$, we get $p_2 \in F$. Since F is upward closed, also $q_2 \in F$. Hence, $v.u \in L(U) = L$ as desired.

⇐ Consider a language $L \subseteq \Sigma^*$ whose Nerode quasi order $\leq_L$ is a WQO. We define the trivial detWSTS $U_L = (\Sigma^*, \leq_L, \varepsilon, \Sigma, \delta, L)$. The states are all words ordered by the Nerode quasi order. The empty word is the initial state, the language L is the set of final states. Note that L is upward closed wrt. $\leq_L$. The transition relation is defined as expected, $\delta(w, a) = w.a$. It is readily checked that $L(U_L) = L$. ◀

The lemma gives a hint on how to construct the witness language T: we should make sure the associated Nerode quasi order $\leq_T$ has an infinite antichain (then it cannot be a WQO). To obtain such an antichain, remember that T will be accepted by a WSTS over the Rado

WQO $(R, \leq_R)$ [38]. It is known that $(\mathbb{D}(R), \subseteq)$ has an infinite antichain. Our strategy for the definition of T will therefore be to translate the infinite antichain in $(\mathbb{D}(R), \subseteq)$ into an infinite antichain in $(\Sigma^*, \leq_T)$. We turn to the details, starting with the Rado WQO.

4.2 Witness Language

Rado Order. Our presentation of the Rado WQO [38] follows [35]. The *Rado set* is the upper diagonal, $R = \{(c, r) \mid c < r\} \subseteq \mathbb{N}^2$. The Rado WQO $\leq_R \subseteq R \times R$ is defined by:

$$(c_1, r_1) \leq_R (c_2, r_2), \qquad \text{if} \qquad r_1 \leq c_2 \ \vee \ (c_1 = c_2 \ \wedge \ r_1 \leq r_2)\ .$$

Given an element (c, r), we call c the *column* and r the *row*, as suggested by Figure 1(left). Columns will play an important role and we denote column i by $C_i = \{(i, r) \mid i < r\} \subseteq R$. To arrive at a larger element in the Rado WQO, one can increase the row while remaining in the same column, or move to the rightmost column of the current row, and select an element to the right, Figure 1(middle).

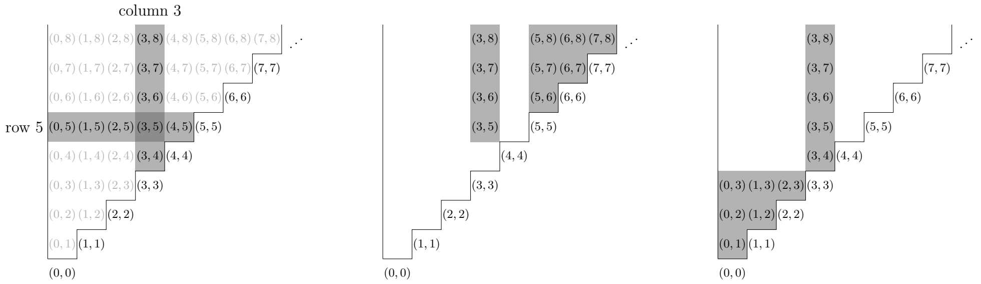


Figure 1 Rado order with the column and row of $(3, 5)$ marked (left), with the elements larger than $(3, 5)$ marked (middle), and with the downward closure of column 3 marked (right).

It is not difficult to see that $(R, \leq_R)$ is a WQO [38]. In an infinite sequence, either the columns eventually plateau out, in which case the rows lead to comparable elements, or the columns grow unboundedly, in which case they eventually exceed the row in the initial pair. The interest in the Rado WQO is that the WQO property is lost when moving to $(\mathbb{D}(R), \subseteq)$. This failure is due to the following well-known fact.

▶ **Lemma 23** ([22], Proposition 4.2)**.** *$\{\downarrow C_i \mid i \in \mathbb{N}\}$ is an infinite antichain in $(\mathbb{D}(R), \subseteq)$.*

To see the lemma, we illustrate the downward closure of a column in Figure 1(right). Inclusion fails to be a WQO as each column C_i forms an infinite set that the downward closure $\downarrow C_j$ with $j > i$ cannot cover. Indeed, $\downarrow C_j$ only has the triangle to the bottom-left of column C_j available to cover C_i, and the triangle is a finite set. We will use exactly this difference between infinite and finite sets in our witness language. It will become clearer as we proceed.

Definition of T. The witness language is the language accepted by $U_R = (R, \leq_R, C_0, \Sigma, \delta, R)$. The set of states is the Rado set, the set of initial states is the first column, and the set of final states is again the entire Rado set. The latter means that a word is accepted as long as it admits a run. The letters in $\Sigma = \{a, \bar{a}, \mathit{zero}\}$ reflect the operation that the transitions $\delta \subseteq R \times \Sigma \times R$ perform on the states:

$$\begin{aligned} \delta((c, r), a) &= (c+1, r+1) & \qquad \delta((c+1, r+1), \bar{a}) &= (c, r) \\ \delta((c+1, r), \mathit{zero}) &= (0, c) & \qquad \delta((0, r+1), \mathit{zero}) &= (0, r)\ . \end{aligned}$$

We will explain the transitions in a moment, but remark that they are designed in a way that makes $\leq_R$ a simulation relation and hence U_R a WSTS.

▶ **Lemma 24.** *$T \in L(WSTS)$.*

To develop an intuition to the language, consider

$$T \cap a^*.\bar{a}^*.zero^* \;=\; \{a^n.\bar{a}^n.zero^i \mid n, i \in \mathbb{N}\} \cup \{a^n.\bar{a}^k.zero^i \mid n, k, i \in \mathbb{N}, n - k > i\}\ .$$

Until reading the first *zero* symbol, the language keeps track of the (Dyck) balance of a and $\bar{a}$ symbols in a word. If the balance becomes negative, the word is directly rejected. If the balance is non-negative, it is the task of the *zero* symbols to distinguish a balance of exactly zero from a positive balance. Words with a balance of exactly zero get accepted regardless of how many *zero* symbols follow. Word that have a positive balance of $c > 0$ when reading the first *zero* get rejected after reading c-many *zero* symbols. As we show, this is enough to distinguish words with a balance of $c > 0$ from words with a balance of $d > 0$ for $d \neq c$, and thus obtain infinitely many classes in the Nerode quasi order. We turn to the details.

▶ **Proposition 25.** *$T \notin L(detWSTS)$*

To prove $T \notin L(\text{detWSTS})$, we associate with each column C_i in the Rado WQO the *column language* $L_i = \{w \in \Sigma^* \mid \delta(C_0, w) = C_i\}$. It consists of those words that reach *all* states in C_i from the initial column C_0. The column languages are non-empty.

▶ **Lemma 26.** *$L_i \neq \emptyset$ for all $i \in \mathbb{N}$.*

We start from the entire initial column, meaning $\varepsilon \in L_0$. The transitions labeled by a move from all states in one column to all states in the next column, $L_i.a \subseteq L_{i+1}$. This already proves the lemma. The $\bar{a}$-labeled transitions undo the effect of the a-labeled transitions and decrement the column, $L_{i+1}.\bar{a} \subseteq L_i$. In the initial column, this is impossible, $\delta(C_0, \bar{a}) = \emptyset$. We illustrate the behaviour of a and $\bar{a}$ in Figure 2 (left).

By Lemma 23, the columns form an antichain in $(\mathbb{D}(R), \subseteq)$. The languages L_i translate this antichain into (actually several) antichains of the form we need. Combined, Lemmas 26, 27, and 22 conclude the proof of Proposition 25, and therefore Theorem 21.

▶ **Lemma 27.** *Every set $K \subseteq \Sigma^*$ with $|K \cap L_i| = 1$ for all $i \in \mathbb{N}$ is an antichain in $(\Sigma^*, \leq_T)$.*

In the rest of the section, we prove Lemma 27. The lemma claims that entire column languages are incomparable in the Nerode quasi order, so we write $L \not\sim_T K$ if for all $w \in L$ and all $v \in K$ we have $w \not\leq_T v$ and $v \not\leq_T w$. Difficult is the incomparability with L_0 stated in the next lemma. The proof will make formal the idea behind the *zero*-labeled transitions.

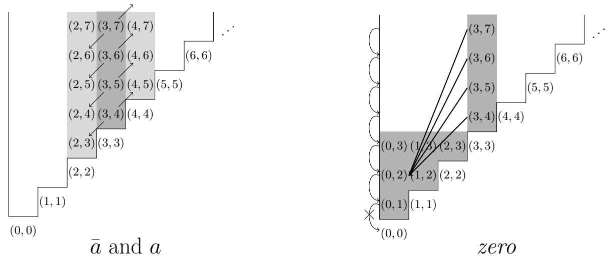


Figure 2 The effect of a and $\bar{a}$-labeled transitions on column 3 (left) and the effect of *zero*-labeled transitions on columns 0 and 3 (right).

▶ **Lemma 28.** *$L_0 \not\sim_T L_k$ for all $k > 0$.*

Proof. Let $w \in L_0$ and $v \in L_k$, meaning w leads to all states in column 0 while v leads to all states in column $k > 0$. It is easy to find a suffix that shows $v \nleq_T w$, namely $\bar{a}$. Appending $\bar{a}$ to v leads to column C_{k-1}, and so $v.\bar{a} \in T$, while there is no transition on $\bar{a}$ from C_0, and so $w.\bar{a} \notin T$.

For $w \nleq_T v$, we need the *zero* transitions. The idea is to make them fail in C_k for $k > 0$, and have no effect in C_0. The problem is that the states in C_k must simulate $(0, r)$ for $r \leq k$. The trick is to fail with a delay. Instead of having no effect in C_0, we let the *zero* transitions decrement the row. Instead of failing in C_k, we let the *zero* transitions imitate the behavior from $(0, k)$ and move to $(0, k-1)$. This is illustrated in Figure 2(right).

By working with column languages, the *zero* transitions fail in C_k with a delay as follows. We have $L_0.\mathit{zero} \subseteq L_0$ but $L_k.\mathit{zero} \nsubseteq L_0$, meaning from C_0 we again reach the entire column C_0, while from C_k we only reach the state $(0, k-1)$. The decrement behavior in the initial column allows us to distinguish the cases by exhausting the rows. Certainly, zero^{k-1} is enabled in large enough states of C_0, meaning $w.\mathit{zero}^k \in T$. The state $(0, k-1)$ reached by $v.\mathit{zero}$, however, does not enable corresponding transitions, $v.\mathit{zero}^k \notin T$. ◀

When executed in C_k with $k > 0$, the *zero* transitions resemble reset transitions [19]. An analogue of leaving C_0 unchanged despite decrements does not exist in the classical model. Moreover, reset nets are defined over $\mathbb{N}^k$ (an ω^2-WQO) as opposed to the Rado set. To conclude the proof of Lemma 27, we lift the previous result to arbitrary column languages.

▶ **Lemma 29.** *$L_i \not\sim_T L_j$ for all $i \neq j$.*

Proof. Let $i < j$ and consider $w \in L_i$ and $v \in L_j$. For $v \nleq_T w$, we append $\bar{a}^j$, which is possible only from the larger column: $v.\bar{a}^j \in T$ but $w.\bar{a}^j \notin T$. For $w \nleq_T v$, we append $\bar{a}^i$. Then $w.\bar{a}^i \in L_0$ while $v.\bar{a}^i \in L_k$ with $k > 0$. Now Lemma 28 applies and yields a suffix u so that $w.\bar{a}^i.u \in T$ but $v.\bar{a}^i.u \notin T$. ◀

The WSTS accepting the witness language T uses non-determinism only in the choice of the initial state. The transitions are deterministic. Moreover, the Rado WQO is embedded in every non-ω^2-WQO [8, Section 2]. Given the determinizability results from [16, Theorem 5], language T thus shows non-determinizability of WSTS with minimal requirements.

5 Downward-Compatible WSTS

We show that the regular separability and non-determinizability results we have obtained for upward-compatible WSTS so far can be lifted to downward-compatible WSTS (DWSTS). In DWSTS, smaller states simulate larger ones and the set of final states is downward closed. We lift our results by establishing general relations between the language classes $L(\text{WSTS})$, $L(\text{DWSTS})$, $L(\text{detWSTS})$, and $L(\text{detDWSTS})$. Figure 3 summarizes them.

$$\begin{tikzcd}
L(\text{detWSTS}) \arrow[r, "{\subsetneq,\ \text{Theorem 21}}"] \arrow[d, leftrightarrow, "{\nsubseteq_{rev},\ \nsupseteq_{rev},\ \text{Lemma 35}} \\ {=_{cmp},\ \text{Lemma 31}}"'] & L(\text{WSTS}) \arrow[d, leftrightarrow, "{=_{rev},\ \text{Lemma 30}}"] \\
L(\text{detDWSTS}) \arrow[r, "{\subsetneq,\ \text{Theorem 34}}"] & L(\text{DWSTS})
\end{tikzcd}$$

Figure 3 Relations between language classes.

Downward Compatibility. A *downward-compatible LTS* (DLTS) is an LTS $D = (Q, I, \Sigma, \delta, F)$ whose states are equipped with a quasi order $\leq \subseteq Q \times Q$ so that the following holds. The final states are downward closed, $\downarrow F = F$, and $\geq$ is a simulation relation. Recall that this means for all $p_1 \leq q_1$ and for all $q_2 \in \delta(q_1, a)$ there is $p_2 \in \delta(p_1, a)$ with $p_2 \leq q_2$. We denote the class of deterministic DLTS by detDLTS. We use L(DLTS) and L(detDLTS) to refer to the classes of all DLTS resp. detDLTS languages. If $\leq$ is also a WQO, we call D a *downward-compatible WSTS* (DWSTS).

Relations between L(DLTS) and L(ULTS). The languages accepted by DLTS are the reverse of the languages accepted by ULTS, and vice-versa. This is easy to see by reversing the transitions. Let $U = (Q, \leq, I, \Sigma, \delta, F)$ be an ULTS. We define $U^{rev} = (Q, \leq, F, \Sigma, \delta^{rev}, \downarrow I)$ to be its reversal. The initial and final states are swapped and the direction of the transitions is flipped, $\delta^{rev} = \{(p, a, q) \mid (q, a, p') \in \delta, p \leq p'\}$. Note that we close the initial states downwards and add transitions from states smaller than the original target. This corresponds to the assumption that the original transitions relate downward-closed sets. The construction can also be applied in reverse to get an ULTS D^{rev} from a DLTS D.

▶ **Lemma 30.** *If $U \in$ ULTS (WSTS), then $U^{rev} \in$ DLTS (DWSTS) and $L(U^{rev}) = L(U)^{rev}$. If $D \in$ DLTS (DWSTS), then $D^{rev} \in$ ULTS (WSTS) and $L(D^{rev}) = L(D)^{rev}$.*

The detDLTS languages are precisely the complements of the detULTS languages. For a detULTS or detDLTS $U = (Q, \leq, y, \Sigma, \delta, F)$, we define the complement $\overline{U} = (Q, \leq, y, \Sigma, \delta, \overline{F})$ by complementing the set of final states [37, Theorem 5].

▶ **Lemma 31.** *$U \in$ detULTS (detWSTS) iff $\overline{U} \in$ detDLTS (detDWSTS), and $L(\overline{U}) = \overline{L(U)}$.*

Behind this is the observation that, under determinism, $\leq$ is a simulation if and only if $\geq$ is [36, Theorem 3.3(ii)]. The details are in the full version [32].

5.1 Lifting Results

Regular Separability of DWSTS. We obtain the regular separability of disjoint DWSTS languages as a consequence of the previous results. More precisely, we need Lemma 30, Theorem 3, and the closure of the regular languages under reversal.

▶ **Theorem 32.** *Let $L_1, L_2 \in L$(DWSTS). We have $L_1 \mid L_2$ if and only if $L_1 \cap L_2 = \emptyset$.*

Non-Determinizability of DWSTS. To show that DWSTS cannot be determinized, recall our witness language T from Section 4. Surprisingly, we have the following.

▶ **Lemma 33.** *$T^{rev} \in L$(detDWSTS) and $\overline{T}^{rev} \in L$(detWSTS).*

For the first claim, recall that the witness language is accepted by the WSTS U_R. The DWSTS ${U_R}^{rev}$ has one minimal initial state, and transition images $\delta^{rev}(p, b)$ with one minimal element for all $p \in R$ and $b \in \Sigma$. Removing simulated states yields a deterministic DWSTS. The details are in the full version [32]. For the second claim, $\overline{T^{rev}} \in L$(detWSTS) by Lemma 31. But $\overline{T^{rev}} = \overline{T}^{rev}$, and so $\overline{T}^{rev} \in L$(detWSTS). Behind this is the fact that bijections commute with complements, and reversal is a bijection.

The lemma allows us to prove non-determinizability for DWSTS. Notably, we do not need a characterization for the languages of deterministic DWSTS.

▶ **Theorem 34.** *$\overline{T} \in L$(DWSTS) $\setminus$ L(detDWSTS) and so L(DWSTS) $\neq L$(detDWSTS).*

Proof. By Lemma 33, $\overline{T}^{rev} \in L$(detWSTS). Lemma 30 yields $\overline{T} \in L$(DWSTS). Suppose $\overline{T} \in L$(detDWSTS). Then $T \in L$(detWSTS) by Lemma 31. This contradicts Proposition 25. ◀

5.2 Consequences

We have shown that neither upward- nor downward-compatible WSTS can be determinized. This does not yet rule out the possibility of determinizing an upward-compatible WSTS into a downward-compatible one, and vice versa. Given the correspondence in Lemma 30, we should allow the determinization to reverse the language. We now show that also this form of reverse-determinization is impossible: there are even deterministic languages that cannot be reverse-determinized. This is by Lemma 33, Proposition 25, and Theorem 34.

▶ **Lemma 35.** *$T^{rev} \in L(detDWSTS)$ but $T \notin L(detWSTS)$. Similarly, $\overline{T}^{rev} \in L(detWSTS)$ but $\overline{T} \notin L(detDWSTS)$*

After reversal, both witness languages T and $\overline{T}$ can be accepted by a deterministic WSTS. When it comes to separability, this means the results from [16] apply to them. A consequence of Lemma 35, however, is that there are WSTS languages that can neither be determinized nor reverse-determinized. An instance is $K = T.\#.\overline{T}^{rev}$ with # a fresh letter.

▶ **Lemma 36.** *$K \in L(WSTS)$, $K \notin L(detWSTS)$, and $K^{rev} \notin L(detDWSTS)$.*

When considering disjoint $K_1, K_2 \in L(\text{WSTS})$ that can neither be determinized nor reverse-determinized, the separability result from [16] does not apply. Theorem 3 is stronger and yields $K_1 \mid K_2$. The situation is similar for downward-compatible WSTS.

6 Conclusion and Future Work

We have shown that disjoint WSTS languages are always separated by a regular language. This strengthens the popular separability result from [16] by showing that the premise in that work (one language had to be accepted by a deterministic WSTS) is not needed. We have also shown that deterministic WSTS accept a strictly weaker class of languages than their non-deterministic counterparts, meaning the premise was a real restriction.

Behind our separability result is a closure of inductive invariants that adds limits of converging sequences, and the fact that the transition relation is compatible with limits. It would be interesting to formulate this in a topological setting [26, 27]. It would also be interesting to apply our invariant closure in settings where separability does not coincide with intersection emptiness and the complexity is open [6]. Finally, it would be interesting to develop compositional verification technology based on separability.

References

1 P. A. Abdulla, M. F. Atig, V. Dave, and S. Narayanan Krishna. On the separability problem of string constraints. In *CONCUR*, volume 171 of *LIPIcs*, pages 16:1–16:19. Dagstuhl, 2020.

2 P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE, 1996.

3 P. A. Abdulla, G. Delzanno, and L. Van Begin. Comparing the expressive power of well-structured transition systems. In *CSL*, volume 4646 of *LNCS*, pages 99–114. Springer, 2007.

4 P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *LICS*, pages 160–170. IEEE, 1993.

5 G.Rozenberg A.Salomaa, editor. *Handbook of Formal Languages*. Springer, 1997. `doi:10.1007/978-3-642-59136-5`.

6 P. Baumann, R. Meyer, and G. Zetzsche. Regular separability in Büchi VASS. In *STACS*, volume 254 of *LIPIcs*, pages 9:1–9:19. Dagstuhl, 2023.

7 M. Blondin, A. Finkel, and P. McKenzie. Handling infinitely branching WSTS. In *ICALP*, volume 8573 of *LNCS*, pages 13–25. Springer, 2014.

8 P. Jančar. A note on well quasi-orderings for powersets. *IPL*, 72(5):155–160, 1999.

9 E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

10 L. Clemente, W. Czerwiński, S. Lasota, and C. Paperman. Regular separability of Parikh automata. In *ICALP*, volume 80 of *LIPIcs*, pages 117:1–117:13. Dagstuhl, 2017.

11 L. Clemente, W. Czerwiński, S. Lasota, and C. Paperman. Separability of reachability sets of vector addition systems. In *STACS*, volume 66 of *LIPIcs*, pages 24:1–24:14. Dagstuhl, 2017.

12 L. Clemente, S. Lasota, and R. Piórkowski. Timed games and deterministic separability. In *ICALP*, volume 168 of *LIPIcs*, pages 121:1–121:16. Dagstuhl, 2020.

13 P. Cousot. *Principles of abstract interpretation.* MIT Press, 2021.

14 W. Czerwiński, P. Hofman, and G. Zetzsche. Unboundedness problems for languages of vector addition systems. In *ICALP*, volume 107 of *LIPIcs*, pages 119:1–119:15. Dagstuhl, 2018.

15 W. Czerwiński and S. Lasota. Regular separability of one counter automata. In *LICS*, pages 1–12. IEEE, 2017.

16 W. Czerwiński, S. Lasota, R. Meyer, S. Muskalla, K. Narayan Kumar, and P. Saivasan. Regular separability of well-structured transition systems. In *CONCUR*, volume 118 of *LIPIcs*, pages 35:1–35:18. Dagstuhl, 2018.

17 W. Czerwiński, W. Martens, L. van Rooijen, M. Zeitoun, and G. Zetzsche. A characterization for decidable separability by piecewise testable languages. *DMTCS*, 19(4), 2017.

18 W. Czerwiński and G. Zetzsche. An approach to regular separability in vector addition systems. In *LICS*, pages 341–354. ACM, 2020.

19 C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In *ICALP*, pages 103–115. Springer, 1998.

20 A. Finkel. A generalization of the procedure of Karp and Miller to well structured transition systems. In *ICALP*, volume 267 of *LNCS*, pages 499–508. Springer, 1987.

21 A. Finkel and J. Goubault-Larrecq. Forward analysis for WSTS, part II: complete WSTS. In *ICALP*, volume 5556 of *LNCS*, pages 188–199. Springer, 2009.

22 A. Finkel and J. Goubault-Larrecq. Forward analysis for WSTS, part II: complete WSTS. *LMCS*, 8(3), 2012.

23 A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *TCS*, 256(1-2):63–92, 2001.

24 G. Geeraerts, J.-F. Raskin, and L. Van Begin. Well-structured languages. *Acta Informatica*, 44(3-4):249–288, 2007.

25 A. R. Gingras. Convergence lattices. *RMJ*, 6(1):85–104, 1976.

26 J. Goubault-Larrecq. On Noetherian spaces. In *LICS*, pages 453–462. IEEE, 2007.

27 J. Goubault-Larrecq. Noetherian spaces in verification. In *ICALP*, volume 6199 of *LNCS*, pages 2–21. Springer, 2010.

28 J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

29 T. Jech. *Set Theory.* Springer, 2002.

30 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

31 M. Kabil and M. Pouzet. Une extension d'un théorème de P. Jullien sur les âges de mots. *RAIRO Theor. Informatics Appl.*, 26:449–482, 1992.

32 Eren Keskin and Roland Meyer. Separability and non-determinizability of WSTS (full version). *CoRR*, abs/2305.02736, 2023. `doi:10.48550/arXiv.2305.02736`.

33 J. Leroux and S. Schmitz. Demystifying reachability in vector addition systems. In *LICS*, pages 56–67. IEEE, 2015.

34 G. Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6:53–68, December 1976. `doi:10.1007/BF02485815`.

35 E. C. Milner. *Basic WQO- and BQO-Theory*, pages 487–502. Springer, 1985.

36 R. Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489. Kaufmann, 1971.

37 M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

38 R. Rado. Partial well-ordering of sets of vectors. *Mathematika*, 1(2):89–95, 1954.

39 R. S. Thinniyam and G. Zetzsche. Regular separability and intersection emptiness are independent problems. In *FSTTCS*, volume 150 of *LIPIcs*, pages 51:1–51:15. Dagstuhl, 2019.

40 G. Zetzsche. Separability by piecewise testable languages and downward closures beyond subwords. In *LICS*, pages 929–938. ACM, 2018.

Compositional Correctness and Completeness for Symbolic Partial Order Reduction

Åsmund Aqissiaq Arild Kløvstad
University of Oslo, Norway

Eduard Kamburjan
University of Oslo, Norway

Einar Broch Johnsen
University of Oslo, Norway

Abstract

Partial Order Reduction (POR) and Symbolic Execution (SE) are two fundamental abstraction techniques in program analysis. SE is particularly useful as a state abstraction technique for sequential programs, while POR addresses equivalent interleavings in the execution of concurrent programs. Recently, several promising connections between these two approaches have been investigated, which result in *symbolic partial order reduction*: partial order reduction of symbolically executed programs. In this work, we provide *compositional* notions of completeness and correctness for symbolic partial order reduction. We formalize completeness and correctness for (1) abstraction over program states and (2) trace equivalence, such that the abstraction gives rise to a complete and correct SE, the trace equivalence gives rise to a complete and correct POR, and their combination results in complete and correct symbolic partial order reduction. We develop our results for a core parallel imperative programming language and mechanize the proofs in Coq.

2012 ACM Subject Classification Theory of computation → Parallel computing models

Keywords and phrases Symbolic Execution, Coq, Trace Semantics, Partial Order Reduction

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.9

Supplementary Material *Software (Coq proofs)*: `https://doi.org/10.5281/zenodo.8070170`

Funding This work was supported by the Research Council of Norway via *SIRIUS* (237898) and *PeTWIN* (294600).

Acknowledgements The first author would like to thank Yannick Zakowski for help with Coq formatting and Erik Voogd for valuable insights on symbolic semantics.

1 Introduction

Program analyses rely on representing the possible reachable states and traces of a program run efficiently and are commonly accompanied by a correctness theorem (*all representable states and traces are reachable*) and possibly a completeness theorem (*all reachable states and traces are represented*). Explicitly listing all states or traces leads to the "state space explosion", as even for simple programs, the number of possible program states may grow so fast that examining them all explicitly becomes infeasible.

One source of this growth is the domain of data – the number of possible values is very large, even for a single integer. Symbolic execution [7, 18, 19] (SE) mitigates this problem by representing values symbolically, thus covering many possible concrete states at once. SE is utilized to great effect in program analysis [3]. Another source of growth is *concurrency*, as the number of possible interleavings grows exponentially. Partial Order Reduction (POR) is a technique for tackling this explosion by taking advantage of the fact that independent events can be reordered without affecting the final result [16].

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 9; pp. 9:1–9:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The combined use of both POR and SE has recently begun to be investigated [6, 28], called *symbolic partial order reduction (SPOR)*. Notions of correctness and completeness are available for both SE and POR, but how these notions can be composed to obtain correctness and completeness of SPOR remains an open challenge. In this paper, we tackle this challenge and give a compositional notion of correctness and completeness for SPOR, based on the abstraction and equivalence notions that define SE and POR. To formulate such a theory we use *trace*-based semantics. Trace semantics is both expressive [23, 31] and compositional [11], and allows a natural formulation of partial order reduction [6].

$$\begin{tikzcd}
\text{concrete semantics} \arrow[r, harpoon, shift left] \arrow[d, harpoon, shift left, "(1)"'] \arrow[dr, harpoon, shift left] & \text{concrete POR} \arrow[l, harpoon, shift left] \arrow[d, harpoon, shift left] \\
\text{symbolic semantics} \arrow[u, harpoon, shift left] \arrow[r, harpoon, shift left] & \text{symbolic POR} \arrow[l, harpoon, shift left, "(2)"] \arrow[u, harpoon, shift left] \arrow[ul, harpoon, shift left]
\end{tikzcd}$$

Figure 1 State of the art and our contribution.

State of the Art

Figure 1 shows the available correctness and completeness results for SE and for POR. Each corner denotes a program semantics, and the arrows denote correctness and completeness. First, let us examine the left side of the square, which is concerned with SE.

The left edge of Figure 1, labeled (1), is provided by de Boer and Bonsangue [5], who define symbolic and concrete semantics for several minimal imperative languages to formulate and prove notions of correctness and completeness for SE. However, their work is limited to a *sequential setting*. The proof is based on using a suitable abstraction between concrete and symbolic states, that defines the SE.

The bottom edge of Figure 1, labeled (2), is studied by de Boer et al. [6], who formulate partial order reduction for symbolic execution with explicit threads using a syntactic notion of interference freedom and implement this approach in the rewriting logic framework Maude [10]. Their results are not connected to the concrete semantics. The result is based on an equivalence relation between symbolic traces, that defines the SPOR, but does not use an explicit abstraction between states. We discuss further, related results on symbolic execution in Sec. 6.

The top of Figure 1 concerns POR [1, 13, 16, 25] for concrete executions, where numerous implementations are available. The correctness of such a reduction corresponds to the top edge of Figure 1, though it is not usually presented in terms of an equivalence relation as proposed by de Boer et al. Results directly of SPOR are given by Schemmel et al. [28], who apply (dynamic) partial order reduction to symbolic execution using “unfolding” to explore paths. This shows that POR is applicable directly to SE, but does not discuss a generic notion of state abstraction and trace equivalence.

While all four corners of Figure 1 are well established, and several edges have been explored, there exists no general formalization of the properties for state abstraction and trace equivalence needed for a uniform and compositional treatment of different POR algorithms and SE techniques. Hence, the present work unifies notions of correctness and completeness for symbolic execution and partial order reduction, and fills in the remaining (black) edges of Figure 1. By compositional completeness and correctness, we mean that the diagonal follows automatically from the other edges of the figure.

Approach

To fill the gap we formulate concrete and symbolic trace semantics for a small imperative language with parallel composition and show that these semantics enjoy a bisimulation relationship. We then formulate partial order reduction in terms of an equivalence relation on traces, and show that this also leads to a bisimulation of reduced and non-reduced semantics. These bisimulations extend to correctness and completeness results, and compose naturally to semantics with *both* symbolic execution and partial order reduction.

The results are obtained in a framework extending the work of de Boer et al. and are centered around the notions of state abstraction and trace equivalence. Following de Boer et al., state abstraction is given by transforming concrete states according to symbolic states, and a concrete state is abstracted if it can be obtained by some symbolic transformation. Trace equivalence defines an equivalence relation on sequences of events which allows for partial order reduction. In particular, it suffices to explore one trace per equivalence class.

Both symbolic and concrete states are implemented by total functions of variable names with generic properties. To reduce the number of rules and allow for elegant parallel composition the semantics are given by a reduction system in the style of Felleisen and Hieb [12] with contexts formalized as functions on statements and an inductive relation [20]. The full semantics are obtained by stepwise transitive closure, which allows for proofs by induction and case analysis of the final step.

Contributions

Our contribution is threefold.

1. We unify and fill in the remaining edges in the above diagram. In particular we give correctness and completeness relations for concrete partial order reduction, directly relate partial order reduction in the symbolic and concrete case, and compose the results to relate concrete semantics to reduced symbolic semantics.
2. Correctness and completeness for both symbolic execution and partial order reduction are formulated in a parametric fashion, allowing for different implementations of both, providing they fulfill certain conditions.
3. Finally, the entire development is mechanized in Coq [4, 32]. This lends credence to the results and allows for extensions and further work in a systematic manner.

Structure

Section 2 introduces basic notions for symbolic execution with trace semantics for a basic imperative language with parallel composition. Then both concrete and symbolic semantics are given as reduction systems with contexts to handle both sequential and parallel composition. Finally we formulate and prove correctness and completeness of the symbolic semantics with respect to the concrete semantics. Section 3 introduces a notion of trace equivalence that connects correctness and completeness to partial order reduction, which is used in Section 4 to define independence of events in a semantic manner. We then define new PO-reduced semantics for both symbolic and concrete cases, and show that they bisimulate their non-reduced counterparts. Finally, Section 5 connects previous results and shows that bisimulation carries through POR to fill in the upper right half of the diagram. Section 6 and 7 give further related work and concludes.

$$\begin{array}{ll}
e ::= n \mid x \mid e_1 + e_2 & \text{arith. expr.} \\
b ::= \text{true} \mid \text{false} \mid \neg b \mid b_1 \land b_2 \mid e_1 \leq e_2 & \text{bool. expr.} \\
s ::= x := e \mid s_1 \ ; \ s_2 \mid s_1 \parallel s_2 \mid \texttt{if } b \ \{s_1\}\{s_2\} \mid \texttt{while } b \ \{s\} \mid \texttt{skip} & \text{statements}
\end{array}$$

Figure 2 Grammar for expressions and statements.

2 Symbolic Trace Semantics

In this section we introduce the basic notions of our framework. In particular, we define a small imperative language with parallel composition and formulate symbolic and concrete trace semantics for it. We relate the two semantics by a bisimulation defining both trace completeness and trace correctness.

2.1 Basic Notions

For the basic setup we assume a set of program variables *Var*, a set of arithmetic expressions *Aexpr* and a set of Boolean expressions *Bexpr*. Our basic programming language is an imperative language with (side effect free) assignment, conditional branching, iteration and both sequential and parallel composition.

▶ **Definition 2.1** (Syntax). *The sets of arithmetic expressions Aexpr, Boolean expressions Bexpr, and statements Stmt are defined by the grammar in Figure 2, where we let x range over Var, n over $\mathbb{N}$, b over Bexpr, e over Aexpr and s over statements.*

Before we define the semantics, we require a notion of store to express program state. We distinguish between symbolic stores, for symbolic execution, and concrete stores, for concrete execution.

▶ **Definition 2.2** (Symbolic Store). *A symbolic store σ is a* substitution*, i.e., a map from Var to Aexpr denoted by σ.*

We take equality of substitutions to be extensional, that is $\sigma = \sigma'$ if $\sigma(x) = \sigma'(x)$ for all x. An *update* to a substitution is denoted by $\sigma[x := e]$. A substitution can be recursively applied to a Boolean or arithmetic expression, resulting in a new expression. We denote such an application by $e\sigma$.

▶ **Definition 2.3** (Concrete Store). *A concrete store V is a* valuation*, i.e., a map from Var to $\mathbb{N}$ denoted by V.*

Like substitutions, valuations can be updated (denoted $V[x := n]$) and a valuation can be used to evaluate an expression. This evaluation is denoted $V(e)$ and results in a natural number for arithmetic expressions and a Boolean for Boolean expressions. For a Boolean expression b, we say V is a model of b if $V(b) = \text{true}$ and denote this by $V \models b$. The definitions of substitution and evaluation are standard and given in the auxiliary material.

2.2 Trace Semantics

Based on the notion of symbolic and concrete stores, we now give the symbolic and concrete semantics. Both semantics are based on traces, i.e., sequences of *events.* Events are assignments or guards in the symbolic case, or just assignments in the concrete case.

▶ **Definition 2.4** (Symbolic Trace). *A symbolic trace is a sequence of conditions or symbolic assignments defined by the grammar*

$$\tau_S ::= [\,] \mid \tau_S :: (x := e) \mid \tau_S :: b$$

▶ **Definition 2.5** (Concrete Trace). *A concrete trace is a sequence of concrete assignments defined by the grammar*

$$\tau_C ::= [\,] \mid \tau_C :: (x := e)$$

In both cases $[\,]$ denotes the empty trace and we write the trace $[\,] :: x :: y :: z \ldots$ simply as $[x, y, z \ldots]$. The concatenation of τ and τ' is denoted by $\tau \cdot \tau'$. The trace syntax is shared between symbolic and concrete traces, but the difference will be clear from context.

We represent the current program state as a pair of a statement (the program remaining to be executed) and the trace generated so far. Evaluating expressions requires to evaluate the expression in the last substitution or valuation of the trace. To do so, we extract this *final* substitution or valuation from a trace and an initial substitution or valuation by folding over the trace. In the case of a symbolic trace, the result is a symbolic substitution, while a concrete trace results in a concrete valuation.

▶ **Definition 2.6** (Final Substitution). *Given an initial substitution σ, the final substitution of a trace τ_S is denoted $\tau_S \Downarrow_\sigma$ and inductively defined by*

$$\begin{aligned} [\,] \Downarrow_\sigma &= \sigma \\ \tau_S :: b \Downarrow_\sigma &= \tau_S \Downarrow_\sigma \\ \tau_S :: (x := e) \Downarrow_\sigma &= \sigma'[x := (e\sigma')] \textit{ where } \sigma' = \tau_S \Downarrow_\sigma \end{aligned}$$

When $\sigma = id$ we omit it and write $\tau_S \Downarrow$

▶ **Definition 2.7** (Final Valuation). *Given an initial valuation V, the final valuation of a trace τ_C is denoted $\tau_C \Downarrow_V$ and inductively defined by*

$$\begin{aligned} [\,] \Downarrow_V &= V \\ \tau_C :: (x := e) \Downarrow_V &= V'[x := V'(e)] \textit{ where } V' = \tau_C \Downarrow_V \end{aligned}$$

Semantics can then be given by a simple reduction relation on atomic statements (Figure 3), which extends to the full language by S/C-IN-CONTEXT. The symbolic (resp. concrete) relation works on pairs of statements and symbolic (resp. concrete) traces to extend them with appropriate events.

▶ **Definition 2.8** (Symbolic and Concrete Semantics). *The symbolic semantics $\rightarrow$ between two symbolic configurations is given on the left of Fig. 3. The concrete semantics $\Rightarrow$ between two concrete configurations is given on the right of Fig. 3.*

Both semantics are straightforward, we point out three details. First, the main difference is that the rules with branching ($*$-IF-T, $*$-IF-F, $*$-WHILE-T, $*$-WHILE-F) are non-deterministic and add an event in the symbolic case, but are deterministic in the concrete case.

Second, in order to concisely deal with both sequential and parallel composition, we use *contexts* [12]. A context C represents a statement with a "hole" ($\Box$) in it and is generated by the grammar:

$$C ::= \Box \mid (C \,;\, s) \mid (C \parallel s) \mid (s \parallel C)$$

$$\textsc{S-Asgn}\ \frac{}{(\texttt{x} := \texttt{e}, \tau) \rightsquigarrow (\texttt{skip}, \tau :: (x := e))} \qquad \frac{}{(\texttt{x} := \texttt{e}, \tau) \boldsymbol{\rightsquigarrow}_V (\texttt{skip}, \tau :: (x := e))}\ \textsc{C-Asgn}$$

$$\textsc{S-If-T}\ \frac{}{(\texttt{if b } \{\texttt{s}_1\}\{\texttt{s}_2\}, \tau) \rightsquigarrow (\texttt{s}_1, \tau :: b)} \qquad \frac{\tau \Downarrow_V (b) = \text{true}}{(\texttt{if b } \{\texttt{s}_1\}\{\texttt{s}_2\}, \tau) \boldsymbol{\rightsquigarrow}_V (\texttt{s}_1, \tau)}\ \textsc{C-If-T}$$

$$\textsc{S-If-F}\ \frac{}{(\texttt{if b } \{\texttt{s}_1\}\{\texttt{s}_2\}, \tau) \rightsquigarrow (\texttt{s}_2, \tau :: \neg b)} \qquad \frac{\tau \Downarrow_V (b) = \text{false}}{(\texttt{if b } \{\texttt{s}_1\}\{\texttt{s}_2\}, \tau) \boldsymbol{\rightsquigarrow}_V (\texttt{s}_2, \tau)}\ \textsc{C-If-F}$$

$$\textsc{S-While-T}\ \frac{}{(\texttt{while b } \{\texttt{s}\}, \tau) \rightsquigarrow (\texttt{s ; while b } \{\texttt{s}\}, \tau :: b)} \qquad \frac{\tau \Downarrow_V (b) = \text{true}}{(\texttt{while b } \{\texttt{s}\}, \tau) \boldsymbol{\rightsquigarrow}_V (\texttt{s ; while b } \{\texttt{s}\}, \tau)}\ \textsc{C-While-T}$$

$$\textsc{S-While-F}\ \frac{}{(\texttt{while b } \{\texttt{s}\}, \tau) \rightsquigarrow (\texttt{skip}, \tau :: \neg b)} \qquad \frac{\tau \Downarrow_V (b) = \text{false}}{(\texttt{while b } \{\texttt{s}\}, \tau) \boldsymbol{\rightsquigarrow}_V (\texttt{skip}, \tau)}\ \textsc{C-While-F}$$

$$\textsc{S-Seq}\ \frac{}{(\texttt{skip ; s}, \tau) \rightsquigarrow (\texttt{s}, \tau)} \qquad \frac{}{(\texttt{skip ; s}, \tau) \boldsymbol{\rightsquigarrow}_V (\texttt{s}, \tau)}\ \textsc{C-Seq}$$

$$\textsc{S-Par}\ \frac{}{(\texttt{skip} \parallel \texttt{skip}, \tau) \rightsquigarrow (\texttt{skip}, \tau)} \qquad \frac{}{(\texttt{skip} \parallel \texttt{skip}, \tau) \boldsymbol{\rightsquigarrow}_V (\texttt{skip}, \tau)}\ \textsc{C-Par}$$

$$\textsc{S-In-Context}\ \frac{(s, \tau) \rightsquigarrow (s', \tau')}{(C[s], \tau) \to (C[s'], \tau')} \qquad \frac{(s, \tau) \boldsymbol{\rightsquigarrow}_V (s', \tau')}{(C[s], \tau) \Rightarrow_V (C[s'], \tau')}\ \textsc{C-In-Context}$$

Figure 3 Reduction rules for symbolic and concrete semantics.

Intuitively, the statement we are interested in may occur on its own, sequentially before some other statement, or on either side of a parallel operator. By $C[s]$ we denote the statement s in the hole in context C.

Finally, we point out that we model termination by reduction to `skip`.

▶ **Example 2.9.** Consider the program $s = \texttt{y} := \texttt{1} \parallel \texttt{x} := \texttt{3} \parallel \texttt{if X} \leq \texttt{1}\ \{\texttt{Y} := \texttt{2}\}\ \{\texttt{Y} := \texttt{3}\}$. We will show that $(s, [\,]) \to^* (\texttt{skip}, [x := 3, y := 1, x > 1, y := 3])$. In other words that $[x := 3, y := 1, x > 1, y := 3]$ is one possible trace of the program.

First apply S-IN-CONTEXT with $C = \texttt{y} := \texttt{1} \parallel \square \parallel \texttt{if X} \leq \texttt{1}\ \{\texttt{Y} := \texttt{2}\}\ \{\texttt{Y} := \texttt{3}\}$ and S-ASGN to obtain

$$(s, [\,]) \to (\texttt{y} := \texttt{1} \parallel \texttt{skip} \parallel \texttt{if X} \leq \texttt{1}\ \{\texttt{Y} := \texttt{2}\}\ \{\texttt{Y} := \texttt{3}\}, [x := 3])$$

The second assignment is similar, followed by S-IF-F in the context $\texttt{skip} \parallel \texttt{skip} \parallel \square$ to obtain

$$\begin{aligned}&(\texttt{skip} \parallel \texttt{skip} \parallel \texttt{if X} \leq \texttt{1}\ \{\texttt{Y} := \texttt{2}\}\ \{\texttt{Y} := \texttt{3}\}, [x := 3, y := 1])\\ &\qquad\qquad \to (\texttt{skip} \parallel \texttt{skip} \parallel \texttt{Y} := \texttt{3}, [x := 3, y := 1, x > 1])\end{aligned}$$

After the last assignment, the superfluous `skip`s are dispensed with by S-PAR and putting the steps in sequence gives the desired

$$(s, [\,]) \to^* (\texttt{skip}, [x := 2, y := x, z := x])$$

Note that we could choose to apply the contexts in a different order, resulting in five other potential traces.

2.3 Correctness and Completeness

The value of symbolic execution comes from its ability to simultaneously capture many possible concrete execution paths. However, not all of these paths will be feasible for all initial valuations. The feasibility of any particular symbolic trace depends on its *path condition* – a conjunction of guards that allow execution to follow down this particular path – which is computed in a similar fashion to final substitutions.

▶ **Definition 2.10** (Path Condition). *The path condition of a symbolic trace τ_S is denoted $pc(\tau_S)$ and defined by*

$$\begin{aligned} pc([\,]) &= \text{true} \\ pc(\tau_S :: b) &= pc(\tau_S) \wedge b(\tau_S \Downarrow) \\ pc(\tau_S :: (x := e)) &= pc(\tau_S) \end{aligned}$$

Because it is a conjunction of terms, once a path condition becomes false, it cannot become true again. The following lemma captures the contrapositive: a model of a trace's path condition is also a model of any prefix's path condition.

▶ **Lemma 2.11** (Path Condition Monotonicity). *If $V \models pc(\tau :: ev)$, then $V \models pc(\tau)$*

To relate the symbolic and concrete traces we define a notion of abstraction based on the correctness and completeness relations of de Boer and Bonsangue.

▶ **Definition 2.12** (Trace abstraction [5]). *Given an initial valuation V, a symbolic trace τ_S and a concrete trace τ_C we say τ_S V-*abstracts *τ_C if $V \models pc(\tau_S)$ and $\tau_C \Downarrow_V = V \circ \tau_S \Downarrow$*

The steps of the symbolic and concrete systems correspond very closely. Every concrete step corresponds to a symbolic step whose path condition is satisfiable, and every symbolic step with a satisfiable path condition corresponds to a concrete step. In both cases the resulting final states are related by simple composition. This relationship is formalized in the following bisimulation result.

▶ **Theorem 2.13** (Bisimulation). *For any initial valuation V and initial traces τ_0, τ_0' such that τ_0 V-abstracts τ_0':*

- *if there is a concrete step $(s, \tau_0) \Rightarrow_V (s', \tau)$, then there exists a symbolic step $(s, \tau_0') \to (s', \tau')$ such that τ' V-abstracts τ, and*
- *if there is a symbolic step $(s, \tau_0') \to (s', \tau')$ and $V \models pc(\tau')$, then there exists a concrete step $(s, \tau_0) \Rightarrow_V (s', \tau)$ such that $\tau \Downarrow_V = V \circ \tau' \Downarrow$*

By induction over the transitive closure and Lemma 2.11 we obtain correctness and completeness results. Intuitively, correctness means that each symbolic execution whose path condition is satisfied by some initial valuation V corresponds to a concrete execution with the same initial valuation. Additionally its trace abstracts the concrete trace in the sense that the final concrete state is the concretization of V by the final symbolic state. In other words the subset of states described by its path condition contains V, and there is a concrete execution corresponding to the transformation described by its final symbolic state.

▶ **Corollary 2.14** (Trace Correctness). *If $(s, \tau_S) \to^* (s', \tau_S')$, τ_S V-abstracts τ_C, and $V \models pc(\tau_S')$, then there exists a concrete trace τ_C' such that $(s, \tau_C) \Rightarrow_V^* (s', \tau_C')$ and $\tau_C' \Downarrow_V = \tau_C \Downarrow_V \circ (\tau_S' \Downarrow)$.*

Completeness captures the opposite relationship: every concrete execution has a symbolic counterpart. Furthermore the symbolic trace recovers the concrete state, and its path condition is satisfied by the initial valuation.

▶ **Corollary 2.15** (Trace Completeness). *If $(s, \tau_C) \Rightarrow_V^* (s', \tau_C')$ and τ_S V-abstracts τ_C, there exists τ_S' such that $(s, \tau_S) \to^* (s', \tau_S')$ and τ_S' V-abstracts τ_C'.*

3 Trace Equivalence

In this section we introduce a notion of trace equivalence which will be used to formulate partial order reduction in Section 4. Intuitively two traces should be equivalent if execution could continue from either one, i.e., if partial order reduction would prune away one of them.

This is surely the case when their final states are the same. In the symbolic case their path conditions must also be equivalent. Additionally, we do not want to equate traces describing observably different behavior, so equivalent traces must contain the same events. These considerations motivate the following definition.

▶ **Definition 3.1** (Symbolic Trace Equivalence). *Symbolic traces τ and τ' are equivalent (denoted $\tau \sim \tau'$) if*

- *τ' is a permutation of τ,*
- *$\tau \Downarrow_\sigma = \tau' \Downarrow_\sigma$ for all initial substitutions σ, and*
- *$V \models pc(\tau) \iff V \models pc(\tau')$ for all valuations V*

▶ **Definition 3.2** (Concrete Trace Equivalence). *Concrete traces τ and τ' are equivalent (denoted $\tau \simeq \tau'$) if*

- *τ' is a permutation of τ,*
- *$\tau \Downarrow_V = \tau' \Downarrow_V$ for all initial valuations V*

▶ **Example 3.3.** Let $\tau_1 = [y := x, z := x]$ and $\tau_2 = [z := x, y := x]$. It is both the case that $\tau_1 \sim \tau_2$ and $\tau_1 \simeq \tau_2$.[1] They evidently contain the same events and have the same (trivially true) path condition. Any initial substitution σ results in a final substitution $\sigma'(v) = \begin{cases} x, & v \in \{y, z\} \\ \sigma(v), & \text{otherwise} \end{cases}$ and any initial valuation V results in $V'(v) = \begin{cases} V(x), & v \in \{y, z\} \\ V(v), & \text{otherwise} \end{cases}$

Clearly, trace equivalence defines an equivalence relation. Furthermore it allows continued execution in the following sense: given a statement s and a trace τ, we can replace τ with an equivalent trace τ', such that the next execution step will result in two different, but equivalent traces.

▶ **Lemma 3.4** (). *For equivalent traces $\tau \sim \tau'$, if $(s, \tau) \to (s', \tau_1)$ then there exists τ_2 such that $(s, \tau') \to (s', \tau_2)$ and $\tau_1 \sim \tau_2$.*

This lemma also holds for concrete traces with concrete equivalence and reduction system and underlies partial order reduction in both cases.

Crucially, the properties of trace equivalence ensure that it preserves abstraction. The following theorem shows that the notion of V-abstraction carries through trace equivalence, which will allow us to connect it with partial order reduction in the sequel.

▶ **Theorem 3.5** (Abstraction Congruence). *For equivalent symbolic traces $\tau_S \sim \tau'_S$ and concrete traces $\tau_C \simeq \tau'_C$, if τ_S V-abstracts τ_C then τ'_S V-abstracts τ'_C*

▶ **Example 3.6.** Continuing Example 3.3, the symbolic trace τ_1 V-abstracts the concrete trace τ_1 for every V, and so τ_1 also V-abstracts the equivalent concrete trace τ_2.

In fact, every symbolic trace V-abstracts itself viewed as a concrete trace for any V.

[1] Recall that symbolic traces are also concrete traces if they contain no branching events (guards).

3.1 Example: Interference Freedom

The reordering of independent events is the core of many POR approaches. In practice true independence is prohibitively expensive to compute, so some over-approximation is used. Interference freedom is a syntactic over-approximation of independence of events. We show that reordering interference free events is an instance of our notion of trace equivalence.

Interference freedom between ev_1 and ev_2 means that ev_1 does not read or write a variable written by ev_2 and vice versa. Formally:

▶ **Definition 3.7** (Interference Freedom). *Let ev be either a Boolean expression b or an assignment $(x := e)$. $R(ev)$ denotes the set of variables read by ev, ie. all the variables in b or e. $W(ev)$ denotes the set of variables written by ev, ie. x. Then ev_1, ev_2 are* interference free *iff*

$$W(ev_1) \cap W(ev_2) = R(ev_1) \cap W(ev_2) = R(ev_2) \cap W(ev_1) = \emptyset$$

Denote the interference freedom of ev_1 and ev_2 by $ev_1 \diamond ev_2$

Interference freedom is an independence relation in the sense that if $ev_1 \diamond ev_2$, then the final state of $[ev_1, ev_2]$ is equal to that of $[ev_2, ev_1]$. The reason is that interference freedom allows for "simultaneous" updates without worrying about the order of operations in the assignment case, and the variables involved in a Boolean expression can not be changed in the guard case.

On the other hand, interference freedom is an over-approximation which is perhaps most easily seen by events like $(x := x)$ and $(x \leq 3)$. Clearly they are semantically independent since the value of x does not change, but they are not interference free.

Equipped with a concrete independence relation we can construct new traces by reordering adjacent independent events. Such a reordering is captured by the equivalence define above in the sense that it results in an equivalent trace.

▶ **Theorem 3.8** (Interference free reordering is a trace equivalence). *Let $\sim_{IF}$ be the smallest equivalence relation on symbolic traces such that $\tau \cdot [ev_1, ev_2] \cdot \tau' \sim_{IF} \tau \cdot [ev_2, ev_1] \cdot \tau'$ for all τ, τ' and $ev_1 \diamond ev_2$.*

The equivalence relation $\sim_{IF}$ is contained in $\sim$.

The analogous result holds for concrete traces and $\simeq$.

This example shows that a POR scheme based on reordering of independent events is captured by trace equivalence.

4 Correctness and Completeness for Symbolic Partial Order Reduction

We formulate POR in the present setting through the use of trace equivalence (defined above) and use it to define new PO-reduced reduction systems. These new systems bisimulate the non-reduced systems of Section 2, leading directly to correctness and completeness results.

At its core, partial order reduction works by observing that some events commute in the execution of a parallel program. These events can be reordered without affecting the final result, and so it it not necessary to explore *every* interleaving. The reduction is often formulated in terms of an (in)dependence relation that determines which events may be reordered. Such a relation must make sure that independent steps leave the system in equivalent states, regardless of the order they are performed in.

An independence relation lifts to an equivalence relation on traces by permuting adjacent independent events. POR approaches then employ some algorithm to compute the equivalence classes of such a relation and avoid exploring traces in the same class. In practice it is difficult to compute the independence of events, so a sound over-approximation is used instead.

We instead take a more high-level approach. Considering trace equivalence to be a fundamental semantic building block, we develop our POR semantics parametric in this notion. This gives us an abstract notion, independent of the specific algorithm for POR.

To take advantage of partial order reduction, we define new transition systems.

▶ **Definition 4.1** (POR Semantics)**.** *The transition rules for symbolic POR are:*

$$\frac{\tau_0 \sim \tau_0' \qquad (s,\tau_0) \rightsquigarrow (s',\tau)}{(s,\tau_0') \rightsquigarrow_{POR} (s',\tau)} \qquad \frac{(s,\tau) \rightsquigarrow_{POR} (s',\tau')}{(C[s],\tau) \rightarrow_{POR} (C[s'],\tau')}$$

And the transition rules for concrete POR are:

$$\frac{\tau_0 \simeq \tau_0' \qquad (s,\tau_0) \boldsymbol{\rightsquigarrow}_{V} (s',\tau)}{(s,\tau_0') \boldsymbol{\rightsquigarrow}_{\boldsymbol{POR,V}} (s',\tau)} \qquad \frac{(s,\tau) \boldsymbol{\rightsquigarrow}_{\boldsymbol{POR,V}} (s',\tau')}{(C[s],\tau) \Rightarrow_{\boldsymbol{POR,V}} (C[s'],\tau')}$$

This new reduction relation includes the steps of the symbolic case but requires only that the initial trace is *equivalent* in the sense defined in Section 3. Crucially, given a class of equivalent traces we may choose only one of them to continue execution. This is the source of *reduction*. Note that it is possible for (s,τ_0') to be unreachable in the original semantics, however the following completeness and correctness results ensure that this does not affect the final result. This approach most closely resembles *sleep sets* [15, 17] which keeps track of equivalent traces that do not need to be explored.

▶ **Example 4.2.** Consider again the program from Example 2.9 and note that $(y := 1)$ and $(x := 3)$ are independent assignments. In the middle of some computation we are left with $\texttt{skip} \parallel \texttt{skip} \parallel \texttt{if x} \leq \texttt{1}\ \{\texttt{Y} := \texttt{2}\}\{\texttt{Y} := \texttt{3}\}$ and the trace $[x := 3, y := 1]$. However, we have previously explored a computation from the state

$$(\texttt{skip} \parallel \texttt{skip} \parallel \texttt{if x} \leq \texttt{1}\ \{\texttt{Y} := \texttt{2}\}\{\texttt{Y} := \texttt{3}\}, [y := 1, x := 3])$$

Now the POR semantics let us replace the equivalent traces and use this computation instead.

In order to utilize POR, we need to know that the reduced traces still model our programs' behavior. It should not throw away any important traces, nor should it invent new ones by taking unsound equivalence classes. Formally, we want the POR semantics to bisimulate their non-reduced counterpart up to trace equivalence.

▶ **Theorem 4.3** (POR bisimulation)**.** *For equivalent initial traces $\tau_0 \sim \tau_0'$:*

- *If $(s,\tau_0) \rightarrow_{POR} (s',\tau)$ then there exists $(s,\tau_0') \rightarrow (s',\tau')$ such that $\tau \sim \tau'$, and*
- *If $(s,\tau_0) \rightarrow (s',\tau)$ then there exists $(s,\tau_0') \rightarrow_{POR} (s',\tau')$ such that $\tau \sim \tau'$*

For equivalent initial traces $\tau_0 \simeq \tau_0'$ and initial valuation V:

- *If $(s,\tau_0) \Rightarrow_{\boldsymbol{POR,V}} (s',\tau)$ then there exists $(s,\tau_0') \Rightarrow_{V} (s',\tau')$ such that $\tau \simeq \tau'$, and*
- *If $(s,\tau_0) \Rightarrow_{V} (s',\tau)$ then there exists $(s,\tau_0') \Rightarrow_{\boldsymbol{POR,V}} (s',\tau')$ such that $\tau \simeq \tau'$*

From these bisimulation results, correctness and completeness follow by induction. Correctness captures the intuition that every PO-reduced execution corresponds to a non-reduced execution with equivalent final traces. This means that partial order reduction is precise in the sense that it does not introduce new traces with different final states.

Completeness is the opposite relationship: every direct execution has a corresponding reduced execution with equivalent traces. Since equivalent traces result in the same final state, completeness means that we do not lose any possible states when performing partial order reduction.

$$\begin{tikzcd}
(s,[\,]) \Rightarrow^*_V (s',\tau_C) \arrow[r, leftrightarrow, "\textit{Theorem 4.3}"] \arrow[d, leftrightarrow, "\textit{Theorem 2.13}"'] \arrow[dr, leftrightarrow, "\textit{Theorem 5.4}"] & (s,[\,]) \Rightarrow^*_{\boldsymbol{POR,V}} (s',\tau'_C) \arrow[d, leftrightarrow, "\textit{Theorem 5.1}"] \\
(s,[\,]) \rightarrow^* (s',\tau_S) \arrow[r, leftrightarrow, "\textit{Theorem 4.3}"'] & (s,[\,]) \rightarrow^*_{POR} (s',\tau'_S)
\end{tikzcd}$$

$\tau_C \simeq \tau'_C$ $\quad$ $\tau_S \sim \tau'_S$ $\quad$ τ_S V-abstracts τ_C $\quad$ τ'_S V-abstracts τ'_C $\quad$ τ'_S V-abstracts τ_C

Figure 4 Overview of the correctness and completeness results.

▶ **Corollary 4.4** (Correctness and Completeness). *For two equivalent symbolic traces* $\tau_0 \sim \tau'_0$*:*
Completeness *If* $(s,\tau_0) \rightarrow^*_{POR} (s',\tau)$ *then there exists* $(s,\tau'_0) \rightarrow^* (s',\tau')$ *with* $\tau \sim \tau'$
Correctness *If* $(s,\tau_0) \rightarrow^* (s',\tau)$ *then there exists* $(s,\tau'_0) \rightarrow^*_{POR} (s',\tau')$ *with* $\tau \sim \tau'$

For two equivalent concrete traces $\tau_0 \simeq \tau'_0$ *and initial valuation* V*:*
Completeness *If* $(s,\tau_0) \Rightarrow^*_{\boldsymbol{POR,V}} (s',\tau)$ *then there exists* $(s,\tau'_0) \Rightarrow^*_V (s',\tau')$ *with* $\tau \simeq \tau'$
Correctness *If* $(s,\tau_0) \Rightarrow^*_V (s',\tau)$ *then there exists* $(s,\tau'_0) \Rightarrow^*_{\boldsymbol{POR,V}} (s',\tau')$ *with* $\tau \simeq \tau'$

5 Composition of SE and POR

In this section we show that the bisimulation results of Section 2 and 4 compose naturally. We use this composition to fill in the remaining edges of Fig. 1, resulting in Fig. 4. This leads to the main result: a bisimulation relation between direct concrete semantics and symbolic POR semantics. Importantly, this allows reasoning about program analysis using *both* SE and POR with the symbolic trace abstracting the concrete trace.

The results are parametric in abstraction and trace equivalence in the following sense. Any equivalence relation on traces which is contained in ours – that is, whose equivalent traces have equivalent final states and path conditions – can be used to perform partial order reduction. Additionally, any symbolic abstraction satisfying Theorem 3.5 can be used for the symbolic execution. The result is a complete and correct *symbolic partial order reduction* where completeness and correctness follows from the respective completeness and correctness results of SE and POR semantics.

First we relate symbolic and concrete POR by combining Theorem 2.13 and Theorem 4.3.

▶ **Theorem 5.1** (POR-POR Bisimulation). *For initial traces* τ_S, τ_C *such that* τ_S V*-abstracts* τ_C*:*

- *If* $(s,\tau_C) \Rightarrow_{\boldsymbol{POR,V}} (s',\tau'_C)$*, then there exists* $(s,\tau_S) \rightarrow_{POR} (s',\tau'_S)$ *such that* τ'_S V*-abstracts* τ'_C
- *If* $(s,\tau_S) \rightarrow_{POR} (s',\tau'_S)$ *and* $V \models pc(\tau'_S)$*, then there exists* $(s,\tau_C) \Rightarrow_{\boldsymbol{POR,V}} (s',\tau'_C)$ *and* $\tau'_C \Downarrow_V = V \circ (\tau'_S \Downarrow)$

From this bisimulation, correctness and completeness relations are obtained by induction. These results are analogous to the direct relationships in Section 2, which shows that the correctness and completeness of symbolic execution is maintained through partial order reduction. In particular we may work with representatives of an *equivalence class* of traces rather than one single trace – which may greatly reduce the state space – and then perform symbolic execution in this new setting.

▶ **Corollary 5.2** (Trace POR Correctness). *If $(s, \tau_S) \rightarrow^*_{POR} (s', \tau'_S)$, τ_S V-abstracts τ_C, and $V \models pc(\tau'_S)$, then there exists a concrete trace τ'_C s.t $(s, \tau_C) \Rightarrow^*_{\boldsymbol{POR,V}} (s', \tau'_C)$ and $\tau'_C \Downarrow_V = \tau_C \Downarrow_V \circ (\tau'_S \Downarrow)$*

▶ **Corollary 5.3** (Trace POR Completeness). *If $(s, \tau_C) \Rightarrow^*_{\boldsymbol{POR,V}} (s', \tau'_C)$ and τ_S V-abstracts τ_C, there exist τ'_S s.t $(s, \tau_S) \rightarrow^*_{POR} (s', \tau'_S)$ and τ'_S V-abstracts τ'_C.*

We are now ready to state our main result, filling in the diagonal and connecting concrete semantics directly to PO-reduced symbolic semantics. Formally, Theorem 2.13 and Theorem 4.3 can be combined to obtain bisimulation of the basic concrete semantics and PO-reduced symbolic semantics.

▶ **Theorem 5.4** (Total Bisimulation). *For initial traces τ_S, τ_C such that τ_S V-abstracts τ_C:*

- *If $(s, \tau_C) \Rightarrow_V (s', \tau'_C)$, then there exists $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ such that τ'_S V-abstracts τ'_C*
- *If $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ and $V \models pc(\tau'_S)$, then there exists $(s, \tau_C) \Rightarrow_V (s', \tau'_C)$ and $\tau'_C \Downarrow_V = V \circ (\tau'_S \Downarrow)$*

▶ **Corollary 5.5** (Total Correctness). *If $(s, \tau_0) \rightarrow^*_{POR} (s', \tau)$, τ_0 V-abstracts τ'_0 and $V \models pc(\tau)$, then there exists τ' such that $(s, \tau'_0) \Rightarrow^*_V (s', \tau')$ and τ V-abstracts τ'.*

▶ **Corollary 5.6** (Total Completeness). *If $(s, \tau_0) \Rightarrow^*_V (s', \tau)$ and τ'_0 V-abstracts τ_0, there exist τ' s.t $(s, \tau'_0) \rightarrow^*_{POR} (s', \tau')$ and τ' V-abstracts τ.*

Figure 4 shows all four reduction systems – symbolic and concrete, with and without POR. Each double arrow denotes a notion of bisimulation, and we obtain the properties shown: both symbolic and concrete traces are equivalent across POR, and V-abstraction is maintained across the symbolic/concrete divide as well as their composition. Additionally we show the relationships between the four traces – the symbolic traces abstract their concrete counterparts, and the POR traces are equivalent – although by Theorem 3.5 it suffices to know the equivalences and one of the abstractions.

5.1 Discussion

The bisimulations compose naturally. As an example, consider Theorem 5.4 which is obtained by composing the symbolic/concrete bisimulation of Theorem 2.13 and the direct/reduced bisimulation of Theorem 4.3. Starting with a concrete execution with trace τ_C we first obtain a symbolic execution with trace τ_S such that τ_S V-abstracts τ_C. Then the POR-bisimulation of Theorem 4.3 gives a symbolic POR-computation with an equivalent trace τ_S. Since trace equivalence is a congruence for abstraction (Theorem 3.5) and τ_C is equivalent to itself, this final trace also abstracts τ_C.

The ease of this composition is not unexpected, since both abstraction and trace equivalence were explicitly formulated to preserve the relevant parts of the program state. The result is that any partial order reduction which picks equivalent traces in this sense preserves the correctness and completeness properties of the symbolic execution. Explicitly, if the notion of trace equivalence is contained in ours and the symbolic abstraction can be transported along this equivalence in the sense of Theorem 3.5 then the techniques can be composed.

5.2 Mechanization

In this section we cover some of the details of the mechanization in Coq.

The basic building blocks of program state are simple. Both substitutions and valuations are implemented as total maps from strings, parameterized by a result type. Updates, notation and several useful lemmas about maps can be proven generically and the notation mirrors that of Pierce et al. [24]. Similarly traces are an inductive type, parametric in the type of events. In essence they are lists, but extended to the right for convenience, with the expected operations and properties.

Trace *equivalence* is defined as a relation. Then we show necessary properties of this relation, in particular Lemma 3.4 and Theorem 3.5 which are used in proofs. Additionally, we implement an equivalence by permuting independent events and show that it satisfies the same properties if the independence relation does. This part is parametric in the independence relation and serves as an example of a POR relation. The example at the end of Section 3 is an instance with interference freedom as the independence relation 🐓.

Expressions (both arithmetic and Boolean) and statements are inductive types. As an example, the type of statements is given by:

```
Inductive Stmt : Type :=
| SAsgn (x:Var) (e:Aexpr)
| SPar (s1 s2:Stmt)
| SIf (b:Bexpr) (s1 s2:Stmt)
...
```

To give semantics to this language, we define a *head reduction* relation and a type of contexts. The head reduction describes the single step reductions for each atomic and how it transforms the current trace. For example an assignment reduces to `skip` and appends the assignment to the current trace. Here `<{_}>` encloses language statements and `Asgn__S x e` represents the symbolic event $(x := e)$.

```
Variant head_red__S: (trace__S * Stmt) → (trace__S * Stmt) → Prop :=
| head_red_asgn__S: ∀t x e,
   head_red__S (t, <{ x := e }>) (t :: Asgn__S x e, SSkip)
   ...
```

Note that **`Variant`** is a version of **`Inductive`** that does not include recursive constructors.

Contexts are implemented as functions `Stmt → Stmt` along with an inductive relation `is_context: (Stmt → Stmt) → Prop` – an approach inspired by Xavier Leroy [20]. This approach allows us to define transition relation semantics parametric in both the type of contexts and the head reduction relation. The following generalizes the ∗-IN-CONTEXT rules for any type of state `X`. In our case, `X` will be a type of traces, but note that `X` appears on the left – this makes the rule amenable to states represented by product types due to the way parentheses associate.

```
Variant context_red
  (is_cont: (Stmt → Stmt) → Prop) (head_red: relation (X * Stmt))
  : relation (X * Stmt) :=
| ctx_red_intro: ∀C x x' s s',
  head_red (x, s) (x', s') → is_context C →
  context_red is_cont head_red (x, C s) (x', C s').
```

Having used `context_red` with the appropriate `is_context` and `head_red` we obtain the full transition relation by stepwise reflexive-transitive closure to the right (`clos_refl_trans_n1`) from the `Relations` library.

The proofs are performed in two steps. Induction on the transition relation leaves us with either a reflexive step or an induction hypothesis and some sequence followed by a step. Then unfolding and dependent destruction (from `Program.Equality`) can be used on the step to unpack `ctx_red_intro` and split on the head reduction rule while remembering the ultimate and penultimate traces.

6 Related Work

We focus on a simple formal model that permits reasoning about symbolic execution and partial order reduction. De Boer and Bonsangue [5] lay the foundations of our work – a symbolic execution model based on transition systems and symbolic substitutions which may be composed with concrete valuations. They do not consider parallelism, but do apply their model to languages with other features including recursive function calls and dynamic object creation. They also explore a kind of trace semantics for the latter extension, but it differs from the semantics considered herein. Extending the current work with more language features, including procedure calls and synchronization tools would be interesting.

SymPaths [6] explores the use of POR for SE in a manner very similar to ours, but does not explicitly compose the correctness and completeness of SE and POR, nor treat the relationship to partial order reduction in the non-symbolic case. Additionally, their treatment of trace equivalence focuses on one specific independence relation while we take a more abstract view.

Other formal approaches to symbolic execution have also been considered in the literature. Steinhöfel [30] focuses on the semantics of the SE system and uses a concretization function to relate sets of symbolic and concrete states. The Gillian platform [14, 22] and related work [27] uses separation logic to construct a SE system that is parametric in the target memory model. Rosu et al. [21, 26, 29] develop reachability logic to present symbolic execution parameterized by the semantics of the target language. These all present alternative approaches to the left edge of Figure 4.

There are also other approaches to partial order reduction. In particular, dynamic or stateless POR (DPOR) [1, 13, 16, 25] avoids exploring equivalent future traces by identifying backtracking points. Additionally the *unfolding* approach explores partial orders more directly as a tree-like event structure [25]. Unfolding has been fruitfully combined with symbolic execution in practice [28].

7 Conclusion

POR and SE are fundamental abstraction techniques in program analysis. SE is particularly useful as a state abstraction technique for sequential programs, while POR addresses equivalent interleavings in the execution of concurrent programs. In this paper, we study the foundations of both techniques based on transition systems and trace semantics, in the context of a core imperative language with parallelism. The formalization provides a unified view of concrete and symbolic semantics with and without partial order reduction. We further formalize correctness and completeness relations for both POR and SE, and compose these relations to study how SE and POR can be combined while preserving correctness and completeness. Our work shows that the framework of correctness and completeness relations between symbolic and concrete transition systems, introduced by de Boer and Bonsangue, extends to parallelism and trace semantics, and provides a natural setting to study formalizations of abstraction techniques for SE, such as POR.

In addition, our formal development of correctness and completeness relations of SE and POR has been fully mechanized using Coq[2]. We believe the mechanization of this framework in Coq can be useful to the community to study further formalizations of abstraction

[2] Provided as supplementary material at `https://github.com/Aqissiaq/symex-formally-formalized` and `https://zenodo.org/record/8070170`

techniques for symbolic execution and their correctness. In particular, in future work, we plan to extend the framework developed in this paper to understand relations between concrete SE frameworks typically used for software testing [9], such as Klee [8], in which states are described using symbolic stores as in this paper, and abstract SE frameworks typically used for deductive verification, such as KeY [2], in which states are described using predicates.

References

1 Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *Proc. 41st Annual Symposium on Principles of Programming Languages (POPL'14)*, pages 373–384. ACM, 2014. `doi:10.1145/2535838.2535845`.

2 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. `doi:10.1007/978-3-319-49812-6`.

3 Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

4 Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, 2013.

5 Frank S de Boer and Marcello Bonsangue. Symbolic execution formally explained. *Formal Aspects of Computing*, 33(4):617–636, 2021.

6 Frank S de Boer, Marcello Bonsangue, Einar Broch Johnsen, Violet Ka I Pun, S Lizeth Tapia Tarifa, and Lars Tveito. SymPaths: Symbolic execution meets partial order reduction. In *Deductive Software Verification: Future Perspectives*, pages 313–338. Springer, 2020.

7 Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In Martin L. Shooman and Raymond T. Yeh, editors, *Proc. International Conference on Reliable Software 1975*, pages 234–245. ACM, 1975. `doi:10.1145/800027.808445`.

8 Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 209–224. USENIX Association, 2008. URL: `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`.

9 Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013. `doi:10.1145/2408776.2408795`.

10 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. `doi:10.1007/978-3-540-71999-1`.

11 Crystal Chang Din, Reiner Hähnle, Ludovic Henrio, Einar Broch Johnsen, Violet Ka I Pun, and Silvia Lizeth Tapia Tarifa. LAGC semantics of concurrent programming languages. *arXiv preprint arXiv:2202.12195*, 2022.

12 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. `doi:10.1016/0304-3975(92)90014-7`.

13 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121. ACM, 2005. `doi:10.1145/1040305.1040315`.

14 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part i: a multi-language platform for symbolic execution. In *Proc. 41st ACM Conference on Programming Language Design and Implementation (PLDI'20)*, pages 927–942, 2020.

15 Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, pages 176–185. Springer, 1991.

16 Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem.* Springer, 1996.

17 Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2:149–164, 1993.

18 Shmuel Katz and Zohar Manna. Towards automatic debugging of programs. *ACM SIGPLAN Notices*, 10(6):143–155, 1975.

19 James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

20 Xavier Leroy. Mechanized semantics. Course materials, 2020. URL: `https://github.com/xavierleroy/cdf-mech-sem`.

21 Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. A generic framework for symbolic execution: A coinductive approach. *Journal of Symbolic Computation*, 80:125–163, 2017. SI: Program Verification. `doi:10.1016/j.jsc.2016.07.012`.

22 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part ii: Real-world verification for JavaScript and C. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 827–850. Springer, 2021.

23 Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. *DAIMI Report Series*, 6(78), July 1977. `doi:10.7146/dpb.v6i78.7691`.

24 Benjamin C Pierce, A Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, A Tolmach, and B Yorgey. Software foundations, volume 2: Programming language foundations. 2017.

25 César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015*, volume 42 of *LIPIcs*, pages 456–469. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. `doi:10.4230/LIPIcs.CONCUR.2015.456`.

26 Grigore Rosu, Andrei Stefanescu, Stefan Ciobâcá, and Brandon M. Moore. One-path reachability logic. In *Proc. 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'13)*, pages 358–367, 2013. `doi:10.1109/LICS.2013.42`.

27 José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *Proc. 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18. ACM, 2018. `doi:10.1145/3236950.3236956`.

28 Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. Symbolic partial-order execution for testing multi-threaded programs. In *International Conference on Computer Aided Verification*, pages 376–400. Springer, 2020.

29 Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuta, Brandon M. Moore, Traian Florin Şerbănută, and Grigore Roşu. All-path reachability logic. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi*, pages 425–440. Springer, 2014.

30 Dominic Steinhöfel. *Abstract execution: automatically proving infinitely many programs.* PhD thesis, Technische Universität Darmstadt, 2020.

31 Dominic Steinhöfel and Reiner Hähnle. The trace modality. In *Dynamic Logic. New Trends and Applications*, pages 124–140. Springer, 2020. `doi:10.1007/978-3-030-38808-9_8`.

32 The Coq Development Team. The Coq proof assistant, September 2022. `doi:10.5281/zenodo.7313584`.

Monus Semantics in Vector Addition Systems with States

Pascal Baumann
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Khushraj Madnani
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Filip Mazowiecki
University of Warsaw, Poland

Georg Zetzsche
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

Vector addition systems with states (VASS) are a popular model for concurrent systems. However, many decision problems have prohibitively high complexity. Therefore, it is sometimes useful to consider overapproximating semantics in which these problems can be decided more efficiently.

We study an overapproximation, called monus semantics, that slightly relaxes the semantics of decrements: A key property of a vector addition systems is that in order to decrement a counter, this counter must have a positive value. In contrast, our semantics allows decrements of zero-valued counters: If such a transition is executed, the counter just remains zero.

It turns out that if only a subset of transitions is used with monus semantics (and the others with classical semantics), then reachability is undecidable. However, we show that if monus semantics is used throughout, reachability remains decidable. In particular, we show that reachability for VASS with monus semantics is as hard as that of classical VASS (i.e. Ackermann-hard), while the zero-reachability and coverability are easier (i.e. EXPSPACE-complete and NP-complete, respectively). We provide a comprehensive account of the complexity of the general reachability problem, reachability of zero configurations, and coverability under monus semantics. We study these problems in general VASS, two-dimensional VASS, and one-dimensional VASS, with unary and binary counter updates.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Vector addition systems, Overapproximation, Reachability, Coverability

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.10

Related Version *Full Version*: https://arxiv.org/abs/2308.14926

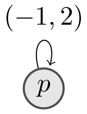
Funding *Filip Mazowiecki*: Supported by the ERC grant INFSYS, agreement no. 950398.

Acknowledgements The authors are grateful to Wojciech Czerwiński and Sylvain Schmitz for discussions, and to Sylvain Schmitz for suggesting the term 'monus'.

1 Introduction

Vector addition systems with states (VASS) are an established model used in formal verification with a wide range of applications, *e.g.* in concurrent systems [22], business processes [39] and others (see the survey [37]). They are finite automata with transitions labeled by vectors over integers in some fixed dimension d. A configuration of a VASS consists of a pair $(p, \mathbf{v})$, denoted $p(\mathbf{v})$, where p is a state and $\mathbf{v}$ is a vector in $\mathbb{N}^d$. As a result of applying a transition labeled by some $\mathbf{z} \in \mathbb{Z}^d$, the vector in the resulting configuration is $\mathbf{v} + \mathbf{z}$. Thus in particular $\mathbf{v} + \mathbf{z} \geq \mathbf{0}$ must hold for the transition to be applicable. The latter requirement is often called the VASS semantics. To avoid ambiguity we will refer to it as the *classical VASS semantics*.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 10; pp. 10:1–10:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



classical	$p(2,0) \rightarrow p(1,2) \rightarrow p(0,4) \not\rightarrow$
integer	$p(2,0) \xrightarrow[\mathbb{Z}]{} p(1,2) \xrightarrow[\mathbb{Z}]{} p(0,4) \xrightarrow[\mathbb{Z}]{} p(-1,6) \xrightarrow[\mathbb{Z}]{*} p(-n,4+2n)$
monus	$p(2,0) \Rightarrow p(1,2) \Rightarrow p(0,4) \Rightarrow p(0,6) \overset{*}{\Rightarrow} p(0,4+2n)$

Figure 1 A VASS in dimension 2 with one state p and one transition t. It has only one transition labeled with $(-1,2)$. We consider possible runs assuming that the initial configuration is $p(2,0)$. We use different notation for steps in each semantics: $\rightarrow$, $\xrightarrow[\mathbb{Z}]{}$, $\Rightarrow$. For the classical semantics ($\rightarrow$) after reaching the configuration $p(0,4)$ the transition can no longer be applied. For the integer semantics ($\xrightarrow[\mathbb{Z}]{*}$) the transition can be applied even in $p(0,4)$, reaching all configurations of the form $p(-n,4+2n)$. Similarly for the monus semantics ($\overset{*}{\Rightarrow}$), but there the configurations reachable from $p(0,4)$ are of the form $p(0,4+2n)$.

The VASS model is also studied with other semantics. One of the most natural variants of VASS semantics is the *integer semantics* (or simply *$\mathbb{Z}$-semantics*), where configurations are of the form $p(\mathbf{v})$, where $\mathbf{v} \in \mathbb{Z}^d$ [25]. There, a transition can always be applied, *i.e.* the resulting configuration is $\mathbf{v}+\mathbf{z}$ and we do not require $\mathbf{v}+\mathbf{z} \geq \mathbf{0}$. In this paper we consider VASS with the *monus semantics*, whose behavior partly resembles both classical and integer semantics. There, a transition can always be applied (as in $\mathbb{Z}$-semantics), however, if as a result the vector in the new configuration would have negative entries, then these are replaced with 0. Thus, vectors in configurations are over the naturals (as in classical semantics). The name monus semantics comes from the monus binary operator, which is a variant of the minus operator.[1] Note that every instance of a VASS can be considered with all three semantics. See Figure 1 for an example.

We study classical decision problems for VASS: reachability and coverability. The input for these problems is a VASS $\mathcal{V}$, an initial configuration $p(\mathbf{v})$, and a final configuration $q(\mathbf{w})$. The *reachability* problem asks whether there is a run from $p(\mathbf{v})$ to $q(\mathbf{w})$. A variant of this problem, called *zero reachability*, requires additionally that in the input the final vector is fixed to $\mathbf{w}=\mathbf{0}$. The *coverability* problem asks whether there is a run from $p(\mathbf{v})$ to $q(\mathbf{w}')$, where $\mathbf{w}' \geq \mathbf{w}$. Note that all three problems can be considered with respect to any of the three VASS semantics. As an example consider the VASS in Figure 1. Then for all three semantics $p(1,2)$ is both reachable and coverable from $p(2,0)$; and $p(0,2)$ is not reachable from $p(2,0)$ (but it is coverable as $(1,2) \geq (0,2)$).

Contribution I: Arbitrary dimension. Our first contribution is settling the complexities of reachability and coverability for VASS with the monus semantics (see Table 1). We prove that reachability is Ackermann-complete by showing that it is inter-reducible with classical VASS reachability, which is known to be Ackermann-complete [30, 9, 29]. This comes as a surprise, since in monus semantics, every transition can always be applied, just like in $\mathbb{Z}$-semantics, where reachability is merely NP-complete [25]. Thus, the monus operation encodes enough information in the resulting configuration that reachability remains extremely hard.

The Ackermann-hardness relies crucially on the fact that the final configuration is non-zero: We also show that the zero reachability problem is EXPSPACE-complete in monus semantics. This uses inter-reducibility with classical VASS coverability, which is EXPSPACE-complete

[1] One can also think that monus semantics is integer semantics, where after every step we apply the ReLU function.

due to seminal results of Lipton and Rackoff [33, 35]. The fact that zero-reachability is significantly easier than general reachability is in contrast to classical semantics, where zero reachability is interreducible with the reachability problem (intuitively, one can modify the input VASS by adding an extra edge that decrements by $\mathbf{w}$).

In another unexpected result, the complexity of coverability drops even more: We prove that it is NP-complete in monus semantics. We complete these results by showing that mixing classical and monus semantics (*i.e.* each transition is designated to either work in classical or monus semantics) makes reachability undecidable.

Contribution II: Fixed dimension. Understanding the complexity of reachability problems in VASS of fixed dimension has received a lot of attention in recent years and is now well understood. This motivates our second contribution: An almost complete complexity analysis of reachability, zero reachability and coverability for VASS with the monus semantics in dimensions 1 and 2. Here, the complexity depends on whether the counter updates are encoded in unary or binary (see Table 1).

We restrict our attention to dimensions 1 and 2 as most research in fixed dimension for the classical semantics. For the classical semantics not much is known about reachability in dimension $d \geq 3$. Essentially, the only known results consist of an upper bound of $\mathbf{F}_7$ that follows from the Ackermann upper bound in the general case [30], and a PSPACE-lower bound that holds already for $d = 2$ [5]. An intuition as to why the jump from 2 to 3 is so difficult is provided already by Hopcroft and Pansiot [27] who prove that the reachability set is always semilinear in dimension 2, and show an example that this is not the case in dimension 3. In contrast, coverability is well understood, and already Rackoff's construction [35] shows that for fixed dimension $d \geq 2$ coverability is in NL and in PSPACE, for unary and binary encoding, respectively (with matching lower bounds [5]).

Key technical ideas. The core insights of our paper are characterizations of the reachability and coverability relations in monus semantics, in terms of reachability and coverability in classical and $\mathbb{Z}$-semantics (Propositions 3.6 and 3.12 and Lemma 3.10). These allow us to apply a range of techniques to reduce reachability problems for one semantics into problems for other semantics, and thereby transfer existing complexity results. There are three cases where we were unable to ascertain the exact complexity: (i) reachability in 2-VASS with unary counter updates, (ii) zero reachability in 1-VASS with binary updates, and (iii) coverability in 1-VASS with binary counter updates. Concerning (i), this is because for 2-VASS with unary updates, it is known that classical reachability is NL-complete [5], but we would need to decide existence of a run that visits intermediate configurations of a certain shape. In the case of 2-VASS with binary updates, the methods from [5] (with a slight extension from [3]) allow this. The other cases, (ii) and (iii), are quite similar to each other. In particular, problem (ii) is logspace-interreducible with classical coverability in 1-VASS with binary updates, for which only an NL lower bound and an NC^2 upper bound are known [2].

Monus semantics as an overapproximation. Recall the example in Figure 1. Notice that every configuration reachable in the classical semantics is also reachable in the integer and monus semantics. It is not hard to see that this is true for every VASS model. Such semantics are called *overapproximations* of the classical VASS semantics. Overapproximations are a standard technique used in implementations of complex problems, in particular for the VASS model (see the survey [4]). They allow to prune the search space of reachable configurations, based on the observation that if a configuration is not reachable by an overapproximation then it cannot be reachable in the classical semantics. This is the core idea behind efficient implementations both of the coverability problem [15, 6] and the reachability problem [12, 7].

The two most popular overapproximations, integer semantics [25] and continuous semantics [20], behave similarly for both reachability and coverability problems, namely both problems are NP-complete. Note that all of the implementations mentioned above rely on such algorithms in NP as they can be efficiently implemented via SMT solvers. Interestingly, the monus semantics is an efficient overapproximation only for the coverability problem. (As far as we know this is the first study of a VASS overapproximation with this property.) Therefore, it seems to be a promising approach to try to speed up backward search algorithms using monus semantics (in the same vein as [6]). Whether this leads to improvements in practice remains to be seen in future work.

Related work. We discuss related work for VASS in classical semantics. A lot of research is dedicated to reachability for the flat VASS model, *i.e.* a model that does not allow for nested cycles in runs. In dimension 2 decision problems for VASS reduce to flat VASS, which is crucial to obtain the exact complexities [5]. It is known that in dimensions $d \geq 3$ such a reduction is not possible, but this raised natural questions of the complexity for flat VASS in higher dimensions [8, 10]. Another research direction is treating the counters in VASS models asymmetrically. For example, it is known that allowing for zero tests in VASS makes reachability and coverability undecidable (they essentially become Minsky machines). However, it was shown that if only one of the 2 counters is allowed to be zero tested then both reachability and coverability remain PSPACE-complete [31]. A different asymmetric question is when one counter is encoded in binary and the other is encoded in unary. Then recently it was shown that coverability is in NP [34] but it is unknown whether there is a matching lower bound. Finally, there are two important extensions of the VASS model: branching VASS (where runs are trees, not paths), and pushdown VASS (with one pushdown stack). For branching VASS, coverability is 2EXPTIME-complete [11]. The complexity of reachability is well understood in dimension 1 [23, 18] but in dimension 2 or higher it is unknown whether it is decidable. For pushdown VASS only coverability in dimension 1 is known to be decidable [32], otherwise decidability of both reachability and coverability remain open problems. Recently some progress was made on restricted pushdown VASS models [14, 21]. The monus semantics is a natural overapproximation that can be studied in all of these variants. Finally, let us mention that VASS with monus semantics fit into the very general framework of G-nets [13], but does not seem to fall into any of the decidable subclasses studied in [13]. However, if we equip VASS with with the usual well-quasi ordering on configurations, it is easy to see that even with monus semantics, they constitute well-structured transition systems (WSTS) [19, 1], which makes available various algorithmic techniques developed for WSTS.

Organization. In Section 2 we formally define the VASS model and the classical, integer and monus semantics. In Section 3 we prove the results in arbitrary dimension. Then in Section 4 and Section 5 we prove the results in dimension 2 and 1, respectively.

2 Vector addition systems with monus semantics: Main results

Given a vector $\mathbf{v} \in \mathbb{Z}^d$ we write $\mathbf{v}[i]$ for the value in the i-th coordinate, where $i \in \{1, \ldots, d\}$. We also refer to i as the i-th counter and write that it contains $\mathbf{v}[i]$ tokens. Given two vectors $\mathbf{v}$ and $\mathbf{v}'$ we write $\mathbf{v} \geq \mathbf{v}'$ if $\mathbf{v}[i] \geq \mathbf{v}'[i]$ for all $i = 1, \ldots, d$. By $\mathbf{0}^d$ we denote the zero vector in dimension d. We also simply write $\mathbf{0}$ if d is clear from context.

Vector addition systems with states. A vector addition system with states (VASS) is a triple $\mathcal{V} = (d, Q, \Delta)$, where $d \in \mathbb{N}$, Q is a finite set of states and $\Delta \subseteq Q \times \mathbb{Z}^d \times Q$ is a finite set of transitions. Throughout the paper we fix a VASS $\mathcal{V} = (d, Q, \Delta)$.

We start with the formal definitions in the *classical semantics.* A configuration of a VASS is a pair $p(\mathbf{v}) \in Q \times \mathbb{N}^d$, denoted $p(\mathbf{v})$. Any transition $t \in \Delta$ induces a successor (partial) function $\mathsf{Succ}_t : Q \times \mathbb{N}^d \to Q \times \mathbb{N}^d$ such that $\mathsf{Succ}_t(q(\mathbf{v})) = q'(\mathbf{v}')$ iff $t = (q, \mathbf{z}, q')$ and $\mathbf{v}' = \mathbf{v} + \mathbf{z}$. This successor function can be lifted up to Δ to get a step relation $\rightarrow_{\mathcal{V}}$, such that any pair of configuration $C \rightarrow_{\mathcal{V}} C'$ iff there exists $t \in \Delta$ with $\mathsf{Succ}_t(C) = C'$. A *run* is a sequence of configurations

$$q_0(\mathbf{v}_0), q_1(\mathbf{v}_1), q_2(\mathbf{v}_2), \ldots, q_k(\mathbf{v}_k)$$

such that for every $0 < j \leq k$, $q_{j-1}(\mathbf{v}_{j-1}) \rightarrow_V q_j(\mathbf{v}_j)$. If there exists such a run we say that $q_k(\mathbf{v}_k)$ is reachable from $q_0(\mathbf{v}_0)$ and denote it $C_0 \xrightarrow{*}_{\mathcal{V}} C_k$. We call $\xrightarrow{*}_{\mathcal{V}}$ the reachability relation in the classical VASS semantics.

In this paper we consider two additional semantics. The first is called the *integer semantics* (or $\mathbb{Z}$*-semantics*). A configuration in this semantics is a pair $p(\mathbf{v}) \in Q \times \mathbb{Z}^d$ (hence, values of vector coordinates can drop below zero). The definitions of successor function, step relation and run are analogous as for the classical semantics. By $\xrightarrow[\mathbb{Z}]{}_{\mathcal{V}}$ and $\xrightarrow[\mathbb{Z}]{*}_{\mathcal{V}}$, we denote the step and reachability relations in the $\mathbb{Z}$-semantics, respectively.

The second is called *monus semantics.* The configurations are the same as in the classical semantics. The difference is in the successor function. Every transition $t \in \Delta$ induces a successor function $\mathsf{Succ}_t : Q \times \mathbb{N}^d \to Q \times \mathbb{N}^d$ as follows: $\mathsf{Succ}_t(q(\mathbf{v})) = q'(\mathbf{v}')$ iff $t = (q, \mathbf{z}, q')$ and for all $j \in \{1, 2, \ldots d\}$, $\mathbf{v}'[j] = \max(\mathbf{v}[j] + \mathbf{z}[j], 0)$. We write in short $\mathbf{v}' = \max(\mathbf{v} + \mathbf{z}, \mathbf{0})$. Step relation and runs are defined analogously as in the case of classical semantics. By $\Rightarrow_{\mathcal{V}}$ and $\xRightarrow{*}_{\mathcal{V}}$, we denote the step and reachability relations in the monus semantics, respectively.

We drop the subscript $\mathcal{V}$ from the above relations when the VASS is clear from context. We write that a run is a *classical run*, a $\mathbb{Z}$ *run* or a *monus run* to emphasize the considered semantics. An example highlighting the differences between the three semantics is in Figure 1.

Decision problems. We study the following decision problems for VASS.

The classical *reachability problem*:

Given A VASS $\mathcal{V} = (d, Q, \Delta)$ and two configurations $p(\mathbf{v})$ and $q(\mathbf{w})$.

Question Does $p(\mathbf{v}) \xRightarrow{*} q(\mathbf{w})$ hold?

The classical *zero reachability problem*:

Given A VASS $\mathcal{V} = (d, Q, \Delta)$, a configuration $p(\mathbf{v})$ and a state q.

Question Does $(p, \mathbf{v}) \xRightarrow{*} q(\mathbf{0}^d)$ hold?

The classical *coverability problem*:

Given A VASS $\mathcal{V} = (d, Q, \Delta)$ and two configurations $p(\mathbf{v})$ and $q(\mathbf{w})$.

Question Does $p(\mathbf{v}) \xRightarrow{*} q(\mathbf{w}')$ hold for some $\mathbf{w}' \geq \mathbf{w}$?

Similarly, the above problems in $\mathbb{Z}$ and classical semantics are defined by replacing $\xRightarrow{*}$ with $\xrightarrow[\mathbb{Z}]{*}$ and $\xrightarrow{*}$, respectively.

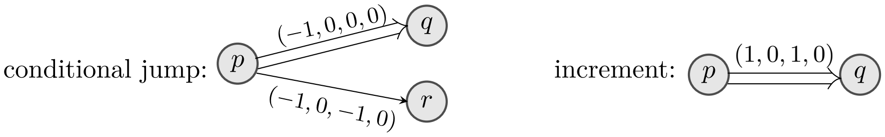


Figure 2 Two gadgets for realizing a zero-testable counter.

Main results. The main complexity results of this work are summarized in Table 1. In Table 2, we recall complexity results for VASS with classical semantics for comparison. We do not split the cases of unary and binary encoding for arbitrary dimensions, since there all lower bounds work for unary, whereas all upper bounds work for binary.

Concerning the *reachability problem*, we note that in all cases where we obtain the exact complexity, it is the same as for the classical VASS semantics. For the other decision problems, there are stark differences: First, while in the classical semantics, *zero reachability* is easily inter-reducible with general reachability, in the monus semantics, its complexity drops in two cases: In 1-VASS with binary counter updates, monus zero reachability is in NC^2 (thus polynomial time), compared to NP in the classical setting. Moreover, in arbitrary dimension, monus zero reachability is EXPSPACE-complete, compared to Ackermann in the classical semantics. For the *coverability problem*, the monus semantics also lowers the complexity in two cases: For binary encoded 2-VASS (NP in monus semantics, PSPACE in classical) and in the general case (NP in monus semantics, EXPSPACE in classical semantics).

Undecidability. To stress the subtle effects of monus semantics, we mention that it leads to undecidability if combined with classical semantics: If one can specify the applied semantics (classical vs. monus) for each transition, then (zero) reachability becomes undecidable.

We sketch the proof using Figure 2. It shows two gadgets, where "→" transitions use classical semantics and "⇒" transitions use monus semantics. The two gadgets realize a counter with zero test: The left gadget is a conditional jump ("if zero, then go to q, otherwise decrement and go to r"), whereas the right gadget is just an increment. In intended runs (i.e. where the left gadget always takes the intended transition), the counter value is stored both in components 1 and 3. (To realize a full two-counter machine, the same gadgets on components 2 and 4 realize the other testable counter.) Thus, initially, all components are zero. Note that if the left gadget always takes the transitions as intended, then the first and third counter will remain equal. If the gadget takes the upper transition when the counter is not actually zero, then the first counter becomes smaller than the third, and will then always stay smaller. Hence, to reach $(0, 0, 0, 0)$, the left gadget must always behave as intended.

However, coverability remains decidable if we can specify the semantics of each transition. Indeed, suppose we order the configurations of a VASS by the usual well-quasi ordering (i.e. the control states have to agree, and the counter values are ordered component-wise). Then it is easy to see that this results in a well-structured transition system (WSTS) [19, 1]. This also implies, e.g. that termination is decidable in this general setting.

3 Arbitrary dimension

In this section, we prove the complexity results concerning VASS with arbitrary dimension. This will include the characterizations of monus reachability, monus zero reachability, and monus coverability in terms of classical and $\mathbb{Z}$-semantics. We begin with some terminology.

Table 1 Complexity results shown in this work.

Dimension & encoding	Monus Reachability	Monus zero reachability	Monus coverability
1-dim, unary	NL-complete	NL-complete	NL-complete
1-dim, binary	NP-complete	in NC^2	in NC^2
2-dim, unary	in PSPACE	NL-complete	NL-complete
2-dim, binary	PSPACE-complete	PSPACE-complete	NP-complete
arbitrary	Ack-complete	EXPSPACE-complete	NP-complete

Table 2 Known complexities for classical VASS semantics, for comparison.

Dimension & encoding	Reachability	Zero reachability	Coverability
1-dim, unary	NL-complete [38]	NL-complete [38]	NL-complete [38]
1-dim, binary	NP-complete [26]	NP-complete [26]	in NC^2 [2]
2-dim, unary	NL-complete [5]	NL-complete [5]	NL-complete [36]
2-dim, binary	PSPACE-complete [5]	PSPACE-complete [5]	PSPACE-complete [5, 36, 16]
arbitrary	Ack-compl. [30, 29, 9]	Ack-compl. [30, 29, 9]	EXPSPACE-compl. [33, 35]

Paths. A sequence of transitions $(p_1, \mathbf{z}_1, q_1), \ldots, (p_k, \mathbf{z}_k, q_k)$ is valid iff $q_i = p_{i+1}$ for every $1 \leq i < k-1$. Furthermore, we say that it is valid from a given configuration $(p, \mathbf{v})$ if $p = p_0$. We call a valid sequence of transitions a *path*.

Given two paths ρ_1 and ρ_2 if the last state of ρ_1 is equal to the first state of ρ_2 then by $\rho = \rho_1\rho_2$ we denote the path defined as the sequence ρ_1 followed by the sequence ρ_2. Similarly, we use this notation with more paths, *e.g.* $\rho = \rho_1\rho_2 \ldots \rho_k$ means that the path ρ is composed from k paths: $\rho_1, \ldots \rho_k$.

Fix a path $\rho = (p_0, \mathbf{z}_0, p_1), \ldots, (p_{k-1}, \mathbf{z}_{k-1}, p_k)$. We say that $\mathbf{z} = \sum_{i=0}^{k-1} \mathbf{z}_i$ is the effect of the path ρ. Notice that while for classical and $\mathbb{Z}$-semantics the effect of a path can be computed by subtracting the vectors in the last and first configurations, this is not necessarily true for monus semantics. In Figure 1 consider the path $\rho = t, t, t$. The effect is $(-3, 6)$. In the $\mathbb{Z}$-semantics $(2, 0) \xrightarrow[\mathbb{Z}]{*} (-1, 6)$ and the difference $(-1, 6) - (2, 0)$ is precisely the effect of ρ. In the monus semantics it is not the case as $(2, 0) \overset{*}{\Longrightarrow} (0, 6)$. This is because a run in monus semantics can lose some decrements, unlike in classical and $\mathbb{Z}$-semantics.

▶ Remark 3.1. Observe that every classical and $\mathbb{Z}$ run defines a unique path from the initial configuration. For monus semantics uniqueness is not guaranteed as it is possible that a run induces more than one path. Indeed, suppose $p(2, 0) \Rightarrow q(1, 0)$. This could be realised by any transition of the form $(p, (-1, z), q)$, where $z \leq 0$. Conversely, a path induces a unique run for $\mathbb{Z}$ and monus semantics. Formally, consider a path $(p_0, \mathbf{z}_1, p_1), \ldots, (p_{k-1}, \mathbf{z}_k, p_k)$ from a configuration $s(\mathbf{v})$. Then, in the $\mathbb{Z}$ and monus semantics there exists a unique corresponding run. In the classical semantics a path might be blocked if a counter drops below zero (see *e.g.* Figure 1). We write $p_0(\mathbf{v}_0) \xrightarrow{\rho} p_k(\mathbf{v}_k)$, $p_0(\mathbf{v}_0) \xrightarrow[\mathbb{Z}]{\rho} p_k(\mathbf{v}_k)$ and $p_0(\mathbf{v}_0) \overset{\rho}{\Longrightarrow} p_k(\mathbf{v}_k)$ if $p_0(\mathbf{v}_0), \ldots, p_k(\mathbf{v}_k)$ is a run in classical, integer and monus semantics, respectively. Recall that for classical and $\mathbb{Z}$-semantics $\mathbf{v}_{i+1} - \mathbf{v}_i = \mathbf{z}_i$, and for monus semantics $\mathbf{v}_{i+1} = \max(\mathbf{v}_i + \mathbf{z}_i, \mathbf{0})$.

Consider a run $R = p_0(\mathbf{v}_0), \ldots, p_k(\mathbf{v}_k)$ (in any semantics). We say that the counter $j \in \{1, \cdots, d\}$ hits 0 iff $\mathbf{v}_i[j] = 0$ for some $1 \leq i \leq k$. Similarly, we say that the counter $j \in \{1, \cdots, d\}$ goes negative in R iff $\mathbf{v}_i[j] < 0$ for some $0 \leq i \leq k$ (this can happen only in the $\mathbb{Z}$-semantics).

Let $\rho = (p_0, \mathbf{z}_0, p_1) \dots (p_{k-1}, \mathbf{z}_{k-1}, p_k)$ be a path such that R is the unique run corresponding to ρ from the initial configuration $p_0(\mathbf{v}_0)$. We say that (ρ, R) or $p_0(\mathbf{v}_0) \overset{\rho}{\Rightarrow} p_k(\mathbf{v}_k)$ is lossy for the counter $j \in \{1, \cdots, d\}$ iff $\mathbf{v}_i[j] - \mathbf{v}_{i-1}[j] \neq \mathbf{z}_{i-1}[j]$ for some $1 \leq i \leq k$ (a lossy run can happen only in the monus semantics).

▶ Remark 3.2. Integer and monus semantics are overapproximations of the classical semantics. That is, $s(\mathbf{v}) \xrightarrow{\rho} t(\mathbf{w})$ implies $s(\mathbf{v}) \underset{\mathbb{Z}}{\xrightarrow{\rho}} t(\mathbf{w})$ and $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w})$. The converse is not always the case (see Figure 1). Moreover, $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w})$ implies $s(\mathbf{v}) \xrightarrow{\rho} t(\mathbf{w})$ if $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w})$ is not lossy. Notice that if in $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w})$, none of the counters $j \in \{1, \dots, d\}$ hits 0 then it is not a lossy run. Similarly, $s(\mathbf{v}) \underset{\mathbb{Z}}{\xrightarrow{\rho}} t(\mathbf{w})$ implies $s(\mathbf{v}) \xrightarrow{\rho} t(\mathbf{w})$ if, in the former run, none of the counters $j \in \{1, \dots, d\}$ goes negative.

Characterizing Monus Reachability. Our first goal is to characterize the reachability problem for the monus semantics in terms of the classical semantics. We start with some propositions that relate monus runs to $\mathbb{Z}$ runs and classical runs. Let ρ be a path and $s_0(\mathbf{v}_0)$ a configuration. Let $s_0(\mathbf{v}_0) \dots s_k(\mathbf{v}_k)$ be the unique $\mathbb{Z}$ run defined by ρ and $s_0(\mathbf{v}_0)$. We define the vector $\mathbf{m} = \min_{\mathbb{Z}}(\rho, s_0, \mathbf{v}_0)$ by $\mathbf{m}[i] = \min(\min_{j=0}^{k} \mathbf{v}_j[i], 0)$. Intuitively, it is the vector of minimal values in the $\mathbb{Z}$ run, but note that $\mathbf{m} \leq \mathbf{0}$.

For the next two propositions we fix a configuration $s_0(\mathbf{v}_0) \in Q \times \mathbb{N}^d$, a path $\rho = (s_0, \mathbf{z}_0, s_1) \dots (s_{k-1}, \mathbf{z}_{k-1}, s_k)$, and $\mathbf{m} = \min_{\mathbb{Z}}(\rho, s_0, \mathbf{v}_0)$.

▶ **Proposition 3.3.** *Consider the unique runs induced by ρ from $s_0(\mathbf{v}_0)$ in $\mathbb{Z}$-semantics*

$$s_0(\mathbf{v}_0), \dots, s_{k-1}(\mathbf{v}_{k-1}), s_k(\mathbf{v}_k),$$

and in monus semantics

$$s_0(\mathbf{v}'_0), \dots, s_{k-1}(\mathbf{v}'_{k-1}), s_k(\mathbf{v}'_k).$$

where $\mathbf{v}'_0 = \mathbf{v}_0$. Then $\mathbf{v}'_k = \mathbf{v}_k - \mathbf{m}$.

Proof (sketch). We analyse the behavior of every counter j. Recall that the $\mathbb{Z}$ run and the monus run have the same value in the counter j until the first time the value of j becomes negative in the $\mathbb{Z}$ run. We denote this as $\mathbf{v}_i[j] = -u$. Note that $\mathbf{v}'_i[j] = 0$. Hence, $\mathbf{v}_i[j] - \mathbf{v}'_i[j] = -u$. It is not hard to see that every time the value of the counter j reaches a new minimum in the $\mathbb{Z}$-semantics, the difference $\mathbf{v}'_i[j] - \mathbf{v}_i[j]$ will be equal to it. We prove this formally by induction on k. Refer to full version for the formal proof. ◀

▶ Remark 3.4. Let $\mathbf{z} \in \mathbb{Z}^d$. A sequence of configurations $s_0(\mathbf{v}_0) \dots s_k(\mathbf{v}_k)$ is a run in $\mathbb{Z}$-semantics corresponding to a path ρ iff $s_0(\mathbf{v}_0 - \mathbf{z}) \dots s_k(\mathbf{v}_k - \mathbf{z})$ is a run in $\mathbb{Z}$-semantics on the same path ρ.

▶ **Proposition 3.5.** *Consider the following unique run corresponding to the path ρ from $s_0(\mathbf{v}_0)$ in the monus semantics*

$$s_0(\mathbf{v}_0), \dots, s_{k-1}(\mathbf{v}_{k-1}), s_k(\mathbf{v}_k).$$

Then the following run, induced by ρ, exists in the classical semantics

$$s_0(\mathbf{v}'_0), \dots, s_{k-1}(\mathbf{v}'_{k-1}), s_k(\mathbf{v}'_k).$$

where $\mathbf{v}'_0 = \mathbf{v}_0 - \mathbf{m}$ and $\mathbf{v}'_k = \mathbf{v}_k$.

Proof. This essentially follows from the definition of $\mathbf{m}$ and Remark 3.4. One just needs to observe that the $\mathbb{Z}$ run with configurations shifted by the vector $-\mathbf{m}$ does not go below zero, hence it is a classical run. See full version for the formal proof. ◀

We now characterize monus reachability in terms of classical reachability.

▶ **Proposition 3.6.** *Let $\mathcal{V} = (d, Q, \Delta)$ be a VASS, let $s(\mathbf{v})$ and $t(\mathbf{w})$ be configurations of $\mathcal{V}$, and let ρ be a path of $\mathcal{V}$. Then, $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w})$ if and only if there is a subset $Z \subseteq \{1, \ldots, d\}$ and a vector $\mathbf{v}' \geq \mathbf{v}$ such that*

1. *$s(\mathbf{v}') \xrightarrow{\rho} t(\mathbf{w})$,*
2. *For every $z \in Z$, the coordinate z hits 0 in $s(\mathbf{v}') \xrightarrow{\rho} t(\mathbf{w})$,*
3. *For every $j \in \{1, \ldots, d\} \setminus Z$, we have $\mathbf{v}'[j] = \mathbf{v}[j]$.*

Proof. ($\implies$) Let $\mathbf{m} = \min_{\mathbb{Z}}(\rho, s, \mathbf{v})$. This direction is implied by Proposition 3.5 along with the following argument. Every counter $j \in \{1, \ldots, d\}$ hits 0 in $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w})$ if and only if it hits 0 in $s(\mathbf{v} - \mathbf{m}) \xrightarrow{\rho} t(\mathbf{w})$. Moreover, if j does not hit 0 in $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w})$ then $\mathbf{m}[j] = 0$.

($\impliedby$) Let $\mathbf{v}' \geq \mathbf{v}$ be a vector as in the statement and let $s(\mathbf{v}') \xrightarrow{\rho} t(\mathbf{w})$. We define $Z \subseteq \{1 \ldots d\}$ such that $i \in Z$ if it hits 0. Moreover, let $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w}'')$. It suffices to show that $\mathbf{w} = \mathbf{w}''$. We write $s(\mathbf{v}') = p_0(\mathbf{v}'_0) \ldots p_k(\mathbf{v}'_k) = t(\mathbf{w})$ and $s(\mathbf{v}) = p_0(\mathbf{v}_0) \ldots p_k(\mathbf{v}_k) = t(\mathbf{w}'')$ for the corresponding runs in the classical and monus semantics, respectively. Note that $\mathbf{v}' \geq \mathbf{v}$ implies $\mathbf{v}'_i \geq \mathbf{v}_i$ for all $0 \leq i \leq k$. By definition of $\mathbf{v}'$ it suffices to consider counters j that hit zero, *i.e.* $\mathbf{v}'_i[j] = 0$ for some $0 \leq i \leq k$. Since $\mathbf{v}'_i \geq \mathbf{v}_i$ we get $\mathbf{v}'_i[j] = 0 = \mathbf{v}_i[j]$. Hence, from i onward both runs agree on the value in counter j. Thus $\mathbf{w} = \mathbf{w}''$. ◀

The reachability problem. We begin with the Ackermann-completeness proof.

▶ **Theorem 3.7.** *Reachability in monus semantics is Ackermann-complete.*

For the upper bound we show how to reduce reachability in monus semantics to reachability in classical semantics. Let $\mathcal{V} = (d, Q, \Delta)$, $s(\mathbf{v})$, and $t(\mathbf{w})$ be the input of the reachability problem in monus semantics. We rely on Proposition 3.6. Intuitively, we have to guess a subset $Z \subseteq \{1, \ldots, d\}$ and a permutation $\sigma\colon [1, k] \to Z$ (where $k = |Z|$). Then we check whether there exists a run as described in Proposition 3.6 with $z_i = \sigma(i)$ for $i \in [1, k]$. To detect the latter run, we construct the VASS $\mathcal{V}_\sigma = (d + k, Q', T')$ as follows. It simulates $\mathcal{V}$, but it has k extra counters to freeze the values of the counter in Z at the points where the coordinates $\sigma(k), \ldots, \sigma(1)$ hit 0 as mentioned in Proposition 3.6.

To remember which counters have already been frozen the set of control states is $Q' = \{q_i \mid q \in Q,\ i \in [0, k]\}$. Intuitively, the index $i \in [0, k]$ stores the information how many counters are frozen. The index i can only increment. Note that guessing the permutation σ allows us to assume that we know the order in which the counters are frozen.

Since we deal with vectors in dimension d and $d + k$ we introduce some helpful notation. We write $\mathbf{e}_j \in \mathbb{Z}^d$ for the unity vector with $\mathbf{e}_j[j] = 1$ and with 0 on other coordinates. Given a vector $\mathbf{z} \in \mathbb{Z}^d$ we define $\mathsf{copy}(\mathbf{z}) \in \mathbb{Z}^{d+k}$ as $\mathsf{copy}(\mathbf{z})[j] = \mathbf{z}[j]$ for $1 \leq j \leq d$ and $\mathsf{copy}(\mathbf{z})[j] = \mathbf{z}[\sigma(j - d)]$ for $d < j \leq d + k$. Intuitively, it simply copies the behaviors of the corresponding counters. We generalise this notation to allow to also remove the effect on some coordinates (*i.e.* "freeze" them). Given $\mathbf{z} \in \mathbb{Z}^d$ and $0 \leq i \leq k$ we define $\mathsf{copy}_i(\mathbf{z}) \in \mathbb{Z}^{d+k}$ as $\mathsf{copy}_i(\mathbf{z})[j] = \mathsf{copy}(\mathbf{z})[j]$ for $1 \leq j \leq d + k - i$ and $\mathsf{copy}_i(\mathbf{z})[j] = 0$ for $d + k - i < j \leq d + k$. In particular $\mathsf{copy}_0(\mathbf{z}) = \mathsf{copy}(\mathbf{z})$ and $\mathsf{copy}_i(\mathbf{z})$ is 0 in the last i counters.

It remains to define the set of transitions T'. In the beginning there are transitions in T' that can arbitrarily increment each counter that belongs to Z and its extra copy: $(s_0, \mathsf{copy}(\mathbf{e}_j), s_0) \in T'$ for every $j \in Z$. Moreover, the counter in the control state can spontaneously be incremented: $(p_i, \mathbf{0}, p_{i+1})$ for every $p \in Q$ and $0 \leq i < k$. For every transition $(p, \mathbf{z}, q) \in T$ and $0 \leq i \leq k$ we define $(p_i, \mathsf{copy}_i(\mathbf{z}), q_i) \in T'$.

The following claim is straightforward by Proposition 3.6:

▷ Claim 3.8. We have $s(\mathbf{v}) \overset{*}{\Rightarrow}_{\mathcal{V}} t(\mathbf{w})$ if and only if there exists a subset $Z \subseteq \{1, \ldots, d\}$ and bijection $\sigma \colon [1, k] \to Z$ such that $s_0(\mathsf{copy}_0(\mathbf{v})) \xrightarrow{*}_{\mathcal{V}_\sigma} t_k(\mathsf{copy}_k(\mathbf{w}))$.

This implies that we can decide monus reachability by guessing a subset $Z \subseteq [1, d]$, guessing a bijection $\sigma \colon [1, k] \to Z$, and deciding reachability in $\mathcal{V}_\sigma$. This yields the upper bound.

For the lower bound we reduce classical reachability to monus reachability. Let $\mathcal{V} = (d, Q, \Delta)$, $s(\mathbf{0})$ and $t(\mathbf{0})$ be the input of the reachability problem in classical semantics (without loss of generality the input vectors can be $\mathbf{0}$). We construct the VASS $\mathcal{V}' = (d + 2, Q', T')$ as follows. The states are $Q' = Q \cup \{t'\}$, where t' is a fresh copy of t.

Again to deal with vectors in different dimension we introduce the following notation. Given $\mathbf{z} \in \mathbb{Z}^d$ we write $\Delta(\mathbf{z}) \in \mathbb{Z}$ for $\Delta(\mathbf{z}) = \sum_{j=1}^{d} \mathbf{z}[j]$, *i.e.* the sum of all components. Based on this we define $\mathsf{extend}(\mathbf{z}) \in \mathbb{Z}^{d+2}$ as: $\mathsf{extend}(\mathbf{z}) = (z, \Delta(z), 0)$ if $\Delta(\mathbf{z}) \geq 0$, and $\mathsf{extend}(\mathbf{z}) = (z, 0, -\Delta(z))$ otherwise.

We define T' as follows. For every $(p, \mathbf{z}, q) \in T$: $(p, \mathsf{extend}(\mathbf{z}), q) \in T'$. Thus, in the $(d + 1)$-th counter, we collect the sum of all non-negative entry sums of the added vectors. Analogously, in the $(d + 2)$-th counter, we collect the sum of all negative entry sums (with a flipped sign). We also add the transition $(t, \mathbf{0}, t') \in T'$, and a "count down" loop: $(t'(\mathbf{0}, -1, -1), t')$, where $(\mathbf{0}, -1, -1)$ is 0 in the first d components and -1 otherwise. The following claim completes the proof of Ackermann-hardness.

▷ Claim 3.9. We have $s(\mathbf{0}, 1, 1) \overset{*}{\Rightarrow} t'(\mathbf{0}, 1, 1)$ in $\mathcal{V}'$ if and only if $s(\mathbf{0}) \xrightarrow{*} t(\mathbf{0})$ in $\mathcal{V}$.

Proof. ($\Longleftarrow$) This is obvious, because every run in classical semantics yields a run in monus semantics between the same configurations.

($\Longrightarrow$) Suppose there is a monus run from $s(\mathbf{0}, 1, 1)$ to $t'(\mathbf{0}, 1, 1)$. Then for some $m \in \mathbb{N}$, there is a transition sequence ρ leading in monus semantics from $s(\mathbf{0}, 1, 1)$ to $t(\mathbf{0}, m, m)$. Now let us execute ρ in $\mathbb{Z}$-semantics. This execution will arrive at some configuration $t(\mathbf{v}, m, m)$ (note that the last two counters are never decreased, except for the final loop). We shall prove that (i) $\mathbf{v} = \mathbf{0}$ and (ii) this execution never drops below zero. First, according to Proposition 3.3, the resulting counter values in monus semantics are always at least the values from $\mathbb{Z}$-semantics. This implies $\mathbf{v} \leq \mathbf{0}$. Next observe that since the right-most components have the same value m, the total sum of all entry sums of added vectors (in the first d entries) must be zero. Thus, $\Delta(\mathbf{v}) = 0$. Together with $\mathbf{v} \leq \mathbf{0}$, this implies $\mathbf{v} = \mathbf{0}$, which shows (i). Second, if the execution in $\mathbb{Z}$-semantics ever drops below zero in some counter i, then by Proposition 3.3 and the fact that in $\mathbb{Z}$-semantics we reach $\mathbf{v} = \mathbf{0}$, this would imply that ρ in monus semantics ends up in a strictly positive value in counter i, which is not true. This shows (ii). Hence, we have shown that the run in $\mathbb{Z}$-semantics is actually a run in classical VASS semantics. Therefore, $s(\mathbf{0}) \xrightarrow{*} t(\mathbf{0})$ in $\mathcal{V}$. ◁

Characterizing zero-reachability. Monus zero-reachability has a simple characterization in terms of classical coverability. Here, $\mathcal{V}^{\mathsf{rev}}$ is obtained by reversing all transitions in $\mathcal{V}$ and their effects. Formally, there is a transition $(p, \mathbf{z}, q)$ in $\mathcal{V}^{\mathsf{rev}}$ iff there is a transition $(q, -\mathbf{z}, p)$ in $\mathcal{V}$.

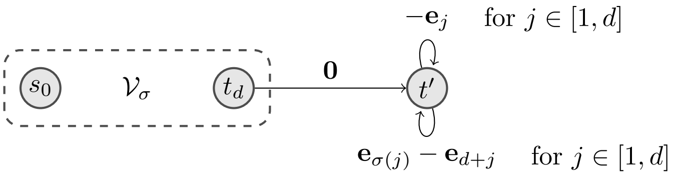

$-\mathbf{e}_j$ for $j \in [1,d]$
s_0
$\mathcal{V}_\sigma$
t_d
$\mathbf{0}$
t'
$\mathbf{e}_{\sigma(j)} - \mathbf{e}_{d+j}$ for $j \in [1,d]$

Figure 3 Construction of $\mathcal{V}'_\sigma$ in reduction from monus coverability to reachability in $\mathbb{Z}$-semantics.

▶ **Lemma 3.10.** *For any* $\mathbf{v}$*, we have* $s(\mathbf{v}) \overset{*}{\Rightarrow}_{\mathcal{V}} t(\mathbf{0})$ *iff* $t(\mathbf{0}) \xrightarrow{*}_{\mathcal{V}^{\text{rev}}} s(\mathbf{v}')$ *for some* $\mathbf{v}' \geq \mathbf{v}$*.*

Proof. By Proposition 3.6, $s(\mathbf{v}) \overset{*}{\Rightarrow} t(\mathbf{0})$ yields a $\mathbf{v}' \geq \mathbf{v}$ with $s(\mathbf{v}') \xrightarrow{*} t(\mathbf{0})$. Conversely, if $s(\mathbf{v}') \xrightarrow{*} t(\mathbf{0})$, then we can pick $Z = [1,d]$ in Proposition 3.6 to obtain $s(\mathbf{v}) \overset{*}{\Rightarrow} t(\mathbf{0})$. ◀

This together with the known complexity of classical coverability [33, 35] immediately implies:

▶ **Proposition 3.11.** *The monus zero-reachability problem is* EXPSPACE*-complete.*

Characterizing coverability. Our third characterization describes coverability in monus semantics in terms of reachability in $\mathbb{Z}$-semantics:

▶ **Proposition 3.12.** *Let* $\mathcal{V} = (d, Q, \Delta)$ *be a VASS and let* $s(\mathbf{v})$ *and* $t(\mathbf{w})$ *be configurations. Then* $s(\mathbf{v}) \overset{*}{\Rightarrow} t(\mathbf{w}'')$ *for some* $\mathbf{w}'' \geq \mathbf{w}$ *if and only if there is a permutation* σ *of* $\{1,\ldots,d\}$ *and* $\mathbb{Z}$*-configurations* $p_d(\mathbf{v}_d), \ldots, p_1(\mathbf{v}_1)$, $t(\mathbf{w}')$ *so that*

1. $s(\mathbf{v}) \xrightarrow[\mathbb{Z}]{*} p_d(\mathbf{v}_d) \xrightarrow[\mathbb{Z}]{*} p_{d-1}(\mathbf{v}_{d-1}) \xrightarrow[\mathbb{Z}]{*} \cdots \xrightarrow[\mathbb{Z}]{*} p_1(\mathbf{v}_1) \xrightarrow[\mathbb{Z}]{*} t(\mathbf{w}')$*,*
2. *for each* $j \in \{1,\ldots,d\}$*, we have* $\mathbf{w}'[j] + |\min(\mathbf{v}_{\sigma^{-1}(j)}[j], 0)| \geq \mathbf{w}[j]$*.*

Proof. ($\Longrightarrow$) Let ρ be any path such that $s(\mathbf{v}) \overset{\rho}{\Rightarrow} t(\mathbf{w}'')$ and $\mathbf{w}'' \geq \mathbf{w}$. Then, by Proposition 3.3 $s(\mathbf{v}) \xrightarrow[\mathbb{Z}]{\rho} t(\mathbf{w}'' + \mathbf{m})$, where $\mathbf{m}$ is the vector of minimum values in the $\mathbb{Z}$ run. The required permutation σ represents the order $\sigma(d), \ldots, \sigma(1)$ in which these coordinates reach their corresponding minimum values. Hence, $s(\mathbf{v}) \xrightarrow[\mathbb{Z}]{\rho} t(\mathbf{w}'' + \mathbf{m})$ is the same as $s(\mathbf{v}) \xrightarrow[\mathbb{Z}]{*} p_d(\mathbf{v}_d) \xrightarrow[\mathbb{Z}]{*} p_{d-1}(\mathbf{v}_{d-1}) \xrightarrow[\mathbb{Z}]{*} \cdots \xrightarrow[\mathbb{Z}]{*} p_1(\mathbf{v}_1) \xrightarrow[\mathbb{Z}]{*} t(\mathbf{w}')$, such that $\mathbf{v}_d[\sigma(d)] = \mathbf{m}[\sigma(d)], \ldots, \mathbf{v}_1[\sigma(1)] = \mathbf{m}[\sigma(1)]$, and $\mathbf{w}''[j] = \mathbf{w}'[j] - \mathbf{m}[j] = \mathbf{w}'[j] + |\mathbf{m}[j]| = \mathbf{w}'[j] + |\min(\mathbf{v}_{\sigma^{-1}(j)}[j], 0)|$ for all $1 \leq j \leq d$. As $\mathbf{w}'' \geq \mathbf{w}$, $\mathbf{w}'[j] + |\min(\mathbf{v}_{\sigma^{-1}(j)}[j], 0)| \geq \mathbf{w}[j]$ for all $1 \leq j \leq d$.

($\Longleftarrow$) This is a direct consequence of Proposition 3.3. It implies that given any permutation σ on $\{1,\ldots,d\}$ and any run $s(\mathbf{v}) \xrightarrow[\mathbb{Z}]{*} p_d(\mathbf{v}_d) \xrightarrow[\mathbb{Z}]{*} p_{d-1}(\mathbf{v}_{d-1}) \xrightarrow[\mathbb{Z}]{*} \cdots \xrightarrow[\mathbb{Z}]{*} p_1(\mathbf{v}_1) \xrightarrow[\mathbb{Z}]{*} t(\mathbf{w}')$ such that $\mathbf{w}'[j] - \min(\mathbf{v}_{\sigma^{-1}(j)}[j], 0) \geq \mathbf{w}[j]$, there is a run from configuration $s(\mathbf{v})$ and reaching a configuration $t(\mathbf{w}'')$ where $\mathbf{w}''[j] = \mathbf{w}'[j] - \mathbf{m}[j] \geq \mathbf{w}'[j] - \min(\mathbf{v}_{\sigma^{-1}(j)}[j], 0) \geq \mathbf{w}[j]$ for all $1 \leq j \leq d$. ◀

We conclude the following.

▶ **Proposition 3.13.** *Monus coverability is* NP*-complete.*

Proof. First we show NP-hardness. In [28, Prop. 5.11], it is shown that it is NP-hard to decide whether a regular language over some alphabet Σ, given as an NFA, contains a word in which every letter appears exactly once. Given such an NFA $\mathcal{A}$ over $\Sigma = \{a_1, \ldots, a_d\}$, we construct a d-VASS $\mathcal{V}$. The VASS $\mathcal{V}$ simulates $\mathcal{A}$ such that when $\mathcal{A}$ reads a_i, $\mathcal{V}$ increments counter i. Moreover, $\mathcal{V}$ maintains a number $k \in \{0, \ldots, d\}$ in its state, which always holds the number of letters read so far. Thus, $\mathcal{V}$ has states q_k, where q is a state of $\mathcal{A}$ and $k \in \{1, \ldots, d\}$. Moreover, let s and t be the initial and final state of $\mathcal{A}$, respectively. Then in $\mathcal{V}$, one can cover $t_d(1, \ldots, 1)$ from $s_0(\mathbf{0})$ in monus semantics if and only if $\mathcal{A}$ accepts some word as above.

We turn to the NP upper bound. Suppose we are given a d-VASS $\mathcal{V} = (d, Q, \Delta)$ and configurations $s(\mathbf{u}), t(\mathbf{v})$. We employ Proposition 3.12. First non-deterministically guess a permutation σ of $[1, d]$. We now construct a $2d$-VASS $\mathcal{V}'_\sigma$ and two configurations c'_1, c'_2 such that in $\mathcal{V}'_\sigma$, we have $c'_1 \xrightarrow[\mathbb{Z}]{*} c'_2$ if and only if there is a run as in Proposition 3.12 with this σ. Since reachability in $\mathbb{Z}$-semantics is NP-complete [25], this yields the upper bound.

Our VASS $\mathcal{V}'_\sigma$ is a slight extension of the VASS $\mathcal{V}_\sigma$ from Theorem 3.7, see Figure 3. Recall that for a permutation $\sigma\colon [1, k] \to Z$, $\mathcal{V}_\sigma$ keeps k extra counters that freeze the values of the counters in Z, in the order $\sigma(k), \sigma(k-1), \ldots, \sigma(1)$. We use this construction, but for our permutation σ of $[1, d]$. Thus, $\mathcal{V}_\sigma$ simulates a run of $\mathcal{V}$ and then freezes the counters $\sigma(d), \ldots, \sigma(1)$ in the extra d counters, in this order. The steps that freeze counters define the vectors $\mathbf{v}_d, \ldots, \mathbf{v}_1$ in Proposition 3.12. Note that for each $\mathbf{v}_i$, only $\mathbf{v}_i[\sigma(i)]$ is important.

To verify the second condition in Proposition 3.12, we introduce an extra state t' and extra transitions as depicted in Figure 3. After executing $\mathcal{V}_\sigma$, $\mathcal{V}'_\sigma$ then has two types of loops: One to move tokens from the counters $d + j$ to counters $\sigma(j)$ (for each $j \in [1, d]$), and one to reduce tokens in counters $1, \ldots, d$. Thus there exists σ such that $s_0(\mathsf{copy}_0(\mathbf{u})) \xrightarrow[\mathbb{Z}]{*} t'(\mathsf{copy}_d(\mathbf{v}))$ in $\mathcal{V}'_\sigma$ if and only if $s(\mathbf{u}) \overset{*}{\Rightarrow} t(\mathbf{v}'')$ for some $\mathbf{v}'' \geq \mathbf{v}$ in $\mathcal{V}$. This proves the NP upper bound. ◀

4 Two-dimensional VASS

In this section we prove the results of Table 1 related to 2-VASS, both for unary and binary encoding. Note that for all three considered problems, reachability, zero reachability, and coverability, we always have an NL lower bound, inherited from state reachability in finite automata. The latter is well-known to be NL-hard, and a VASS without counters (in all considered semantics) is a finite state automaton.

When dealing with binary/unary updates one needs to be careful with the input size. In all problems suppose a VASS $\mathcal{V} = (d, Q, T)$ is in the input. If we are interested in the unary encoding its size is defined as $d + |Q| + \sum_{(p,\mathbf{z},q)\in T} \|\mathbf{z}\|$, where $\|\mathbf{z}\|$ is the absolute value of the maximal coordinate in $\mathbf{z}$. In the binary encoding one needs to change $\|\mathbf{z}\|$ to $\lceil \log(\|\mathbf{z}\| + 1) \rceil$. From this point onwards, we use the term *succinct* VASS for VASS where updates are encoded in binary.

We consider each of the three problems separately.

Reachability. Here we only prove the PSPACE upper bound for monus reachability in binary encoded 2-VASS, which implies the same upper bound for unary encoding. The PSPACE lower bound for binary encoding is inherited from zero reachability, see Proposition 4.3 below.

▶ **Proposition 4.1.** *In succinct 2-VASS, reachability with monus semantics is in* PSPACE.

According to Proposition 3.6, reachability with monus semantics is equivalent to existence of a run under classical semantics, where said run is subject to some additional constraints. Recall that *Presburger arithmetic* is the first-order theory of $(\mathbb{N}, +, <, 0, 1)$. We observe that all the additional constraints of Proposition 3.6 can be expressed by quantifier-free Presburger formulas. This leads us to the so-called *constrained runs problem* for succinct 2-VASS, which was recently shown to be in PSPACE [3], following the fact that classical reachability itself is PSPACE-complete for succinct 2-VASS [5].

Formally, the *constrained runs problem* for succinct 2-VASS is the following:

Given A succinct 2-VASS $\mathcal{V}$, a number $m \in \mathbb{N}$, states $q_1, \dots, q_m$ in $\mathcal{V}$, a quantifier-free Presburger formula $\psi(x_1, y_1, \dots, x_m, y_m)$, and numbers $s, t \in [1, m]$ with $s \leq t$.

Question Does there exist a run $q_0(0,0) \xrightarrow{*} q_1(x_1, y_1) \xrightarrow{*} \cdots \xrightarrow{*} q_m(x_m, y_m)$ that visits a final state between $q_s(x_s, y_s)$ and $q_t(x_t, y_t)$ and satisfies $\psi(x_1, y_1, \dots, x_m, y_m)$?

▶ **Lemma 4.2** ([3, Prop. 6.5]). *The constrained runs problem for succinct 2-VASS is in* PSPACE.

We can now prove Proposition 4.1 by reducing to the constrained runs problem: Let $\mathcal{V}$ be a 2-VASS with configurations $s(\mathbf{v})$ and $t(\mathbf{w})$. According to Proposition 3.6, existence of a run $s(\mathbf{v}) \xRightarrow{*} t(\mathbf{w})$ is equivalent to existence of states p_1, p_2 and a set $Z \subseteq [1, 2]$ such that a run $s(\mathbf{v}') \xrightarrow{*} t(\mathbf{w})$ with $\mathbf{v}' \geq \mathbf{v}$ that is subject to additional requirements enforced by conditions (2) and (3) of the Proposition 3.6. Our PSPACE algorithm enumerates all possibilities of p_1, p_2 and Z, constructing an instance of the constrained run problem each time, and checking for a constrained run in PSPACE using Lemma 4.2. If such a run exists in at least one of the instances, the algorithm accepts, otherwise it rejects. To construct each instance the algorithm first modifies $\mathcal{V}$ to ensure that a starting configuration $s(\mathbf{v}')$ is reachable for any $\mathbf{v}' \geq \mathbf{v}$. To this end a new initial state q_0 is added, with two loops that increment one of the counters each, and a transition that goes to s by adding $\mathbf{v}$. Then the additional requirements of Proposition 3.6 are encoded in quantifier-free Presburger arithmetic, as required by the constrained run problem. Clearly the constructed algorithm runs in PSPACE and decides $s(\mathbf{v}) \xRightarrow{*} t(\mathbf{w})$. For more details refer the full version.

Zero reachability.

▶ **Proposition 4.3.** *Monus zero reachability in 2-VASS is* PSPACE*-complete under binary encoding and* NL*-complete under unary encoding.*

Proof. This is a simple consequence of monus zero reachability being interreducible with classical coverability: Classical coverability in 2-VASS under binary encoding is PSPACE-complete under binary encoding (in [5, Corollary 3.3], this is deduced from [36, p. 108] and [17, Corollary 10] and NL-complete under unary encoding [36, p. 108].

Let $\mathcal{V}$ be a 2-VASS with configurations $s(\mathbf{v})$ and $t(\mathbf{0})$. Then according to Proposition 3.6, we know that $t(\mathbf{0})$ is monus reachable from $s(\mathbf{v})$ if and only if in $\mathcal{V}^{\mathsf{rev}}$ the configuration $s(\mathbf{v})$ is coverable from $t(\mathbf{0})$ with classical semantics. On the other hand, given configurations $s(\mathbf{v})$ and $t(\mathbf{w})$ of a 2-VASS $\mathcal{V}$, we add a new state s' and transition $(s', \mathbf{v}, s)$ to construct the 2-VASS $\mathcal{V}'$. Then classical coverability of $t(\mathbf{w})$ from $s(\mathbf{v})$ in $\mathcal{V}$ is equivalent to the same from $s'(\mathbf{0})$ in $\mathcal{V}'$. Now applying Proposition 3.6 in reverse, the latter is further equivalent to monus reachability of $s'(\mathbf{0})$ from $t(\mathbf{w})$ in $\mathcal{V}'^{\mathsf{rev}}$. ◀

Coverability. By Proposition 3.13, monus coverability is in NP in arbitrary dimension. Thus, it remains to show the NP lower bound.

▶ **Proposition 4.4.** *Monus coverability in succinct 2-VASS is* NP*-hard.*

Proof. We reduce from the *subset sum* problem, which is well-known to be NP-hard. Here, we are given binary encoded numbers $a_1, \dots, a_n, a \in \mathbb{N}$ and are asked whether there is a vector $(x_1, \dots, x_n) \in \{0, 1\}^n$ such that $x_1 a_1 + \cdots + x_n a_n = a$. Given such an instance, we construct the 2-VASS in Figure 4. It is clear that we can cover $t(1, 1)$ from $s(0, 0)$ iff the subset-sum instance is positive: Covering 1 in the first counter means our sum is at least a, whereas covering 1 in the second counter means our sum is at most a. ◀

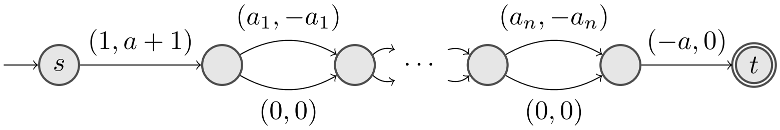


Figure 4 2-VASS to show NP-hardness of coverability in dimension two.

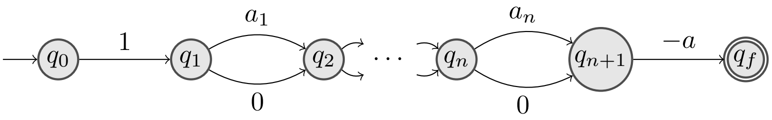


Figure 5 1-VASS to show NP-hardness of monus reachability in dimension one with binary encoded counter updates.

▶ **Proposition 4.5.** *Monus coverability in unary-encoded* 2*-VASS is in* NL*.*

Proof. This follows using the same construction as for Proposition 3.13: Given a 2-VASS, there are only two permutations σ of $\{1, 2\}$. Thus, we can try both permutations σ and construct the VASS $\mathcal{V}'_\sigma$ in logspace. Then, $\mathcal{V}_\sigma$ has dimension $2d$. Thus, we reduce monus coverability in 2-VASS to reachability in $\mathbb{Z}$-semantics in 4-VASS. Since reachability with $\mathbb{Z}$-semantics in each fixed dimension can be decided in NL [24], this provides an NL upper bound. ◀

5 One-dimensional VASS

Reachability. We begin with the proofs regarding reachability.

▶ **Proposition 5.1.** *Monus reachability in* 1*-VASS is in* NL *under unary encoding and in* NP *under binary encoding.*

The proof of Proposition 5.1 relies on the following simple consequence of Proposition 3.6:

▶ **Lemma 5.2.** *Let $\mathcal{V}$ be a* 1*-VASS. Then $s(m) \overset{*}{\Rightarrow}_{\mathcal{V}} t(n)$ if and only if (i) $s(m) \xrightarrow{*}_{\mathcal{V}} t(n)$ or (ii) there exist a state q and number $m' \geq m$ with $s(m') \xrightarrow{*}_{\mathcal{V}} q(0)$ and $q(0) \xrightarrow{*}_{\mathcal{V}} t(n)$.*

For Proposition 5.1, we reduce to reachability in one-counter automata. A *one-counter automaton (OCA)* is a 1-VASS with zero-tests, i.e. special transitions that test the counter for zero instead of adding a number. For encoding purposes, zero tests take up as much space as a transition adding 0 to the counter. In our reduction, the update encoding is preserved: If the input 1-VASS has unary encoding, then the OCA has unary updates as well. If the input 1-VASS has binary updates, then the OCA will too. Then, we can use the fact that in OCA with unary updates, reachability is in NL [38] and for binary updates, it is in NP [26].

The OCA first guesses whether to simulate a run of type (i) or of type (ii) in Lemma 5.2. Then for type (i), it just simulates a classical 1-VASS. For type (ii), it first non-deterministically increments the counter, and then simulates a run of the 1-VASS. However, on the way, it keeps a flag signaling whether the counter has hit 0 at some point (which it can maintain using zero tests). Thus, when simulating runs of type (ii), the OCA only accepts if zero has been hit. For a detailed description, refer to the full version.

▶ **Proposition 5.3.** *Monus reachability in* 1*-VASS is* NP*-hard under binary encoding.*

As in Proposition 4.4, we reduce from subset sum. Given $a_1, \ldots, a_n, a$ in binary, we construct the 1-VASS in Figure 5. Then $q_0(0) \overset{*}{\Rightarrow} q_f(1)$ iff this is a positive instance. Refer to the full version.

Zero reachability and coverability.

▶ **Proposition 5.4.** *Monus zero-reachability in* 1*-VASS is in* NL *under unary encoding and in* NC^2 *under binary encoding.*

Since monus zero-reachability reduces to classical coverability (Lemma 3.10), this follows from existing 1-VASS results: Coverability in 1-VASS is in NL under unary encoding [38] and NC^2 under binary encoding [2].

▶ **Proposition 5.5.** *Monus coverability in* 1*-VASS is in* NL *under unary encoding and in* NC^2 *under binary encoding.*

The first statement follows from Proposition 5.1 and the fact that monus coverability reduces to monus reachability by simply adding a new final state where we can count down. For the NC^2 bound, we use the following consequence of Lemma 3.10 (see the full version).

▶ **Lemma 5.6.** *Let $\mathcal{V}$ be a* 1*-VASS with configurations $s(m)$ and $t(n)$. Then $t(n)$ is monus coverable from $s(m)$ in $\mathcal{V}$ if and only if $t(n)$ is coverable from $s(m)$ in $\mathcal{V}$ under classical semantics or there is a state q of $\mathcal{V}$ such that $t(n)$ is coverable from $q(0)$ in $\mathcal{V}$ under classical semantics and $s(m)$ is coverable from $q(0)$ in $\mathcal{V}^{\mathsf{rev}}$ under classical semantics.*

Proof of Proposition 5.5. It remains to prove the NC^2 upper bound, for which we check the requirements of Lemma 5.6. Let k be the number of states of the input 1-VASS. Observe that Lemma 5.6 yields a logical disjunction over $k+1$ disjuncts, where one disjunct consists of a single coverability check and the remaining k each consist of a logical conjunction over two coverability checks. Classical coverability of binary encoded 1-VASS is in NC^2 [2], and by the definition of this complexity class, we can combine $2k+1$ such checks according to the aforementioned logical relationship and still yield an NC^2-algorithm. Note that this is only possible because k is linear in the size of the input. ◀

References

1 Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 313–321. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561359.

2 Shaull Almagor, Nathann Cohen, Guillermo A. Pérez, Mahsa Shirmohammadi, and James Worrell. Coverability in 1-VASS with Disequality Tests. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 38:1–38:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.38.

3 Pascal Baumann, Roland Meyer, and Georg Zetzsche. Regular Separability in Büchi VASS. In Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté, editors, *40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*, volume 254 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2023.9.

4 Michael Blondin. The ABCs of Petri net reachability relaxations. *ACM SIGLOG News*, 7(3), 2020. doi:10.1145/3436980.3436984.

5 Michael Blondin, Matthias Englert, Alain Finkel, Stefan Göller, Christoph Haase, Ranko Lazic, Pierre McKenzie, and Patrick Totzke. The Reachability Problem for Two-Dimensional Vector Addition Systems with States. *J. ACM*, 68(5):34:1–34:43, 2021. doi:10.1145/3464794.

6 Michael Blondin, Alain Finkel, Christoph Haase, and Serge Haddad. Approaching the coverability problem continuously. In *Proc. 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 480–496, 2016. doi:10.1007/978-3-662-49674-9_28.

7 Michael Blondin, Christoph Haase, and Philip Offtermatt. Directed reachability for infinite-state systems. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2021. doi:10.1007/978-3-030-72013-1_1.

8 Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. Reachability in fixed dimension vector addition systems with states. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 48:1–48:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.48.

9 Wojciech Czerwinski and Lukasz Orlikowski. Reachability in vector addition systems is ackermann-complete. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1229–1240. IEEE, 2021. doi:10.1109/FOCS52979.2021.00120.

10 Wojciech Czerwinski and Lukasz Orlikowski. Lower bounds for the reachability problem in fixed dimensional vasses. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2–5, 2022*, pages 40:1–40:12. ACM, 2022. doi:10.1145/3531130.3533357.

11 Stéphane Demri, Marcin Jurdzinski, Oded Lachish, and Ranko Lazic. The covering and boundedness problems for branching vector addition systems. *J. Comput. Syst. Sci.*, 79(1):23–38, 2013. doi:10.1016/j.jcss.2012.04.002.

12 Alex Dixon and Ranko Lazic. Kreach: A tool for reachability in petri nets. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 405–412. Springer, 2020. doi:10.1007/978-3-030-45190-5_22.

13 Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998. doi:10.1007/BFb0055044.

14 Matthias Englert, Piotr Hofman, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Juliusz Straszynski. A lower bound for the coverability problem in acyclic pushdown VAS. *Inf. Process. Lett.*, 167:106079, 2021. doi:10.1016/j.ipl.2020.106079.

15 Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer, and Filip Nikšić. An SMT-based approach to coverability analysis. In *Proc. 26th International Conference on Computer Aided Verification (CAV)*, pages 603–619, 2014. doi:10.1007/978-3-319-08867-9_40.

16 John Fearnley and Marcin Jurdziński. Reachability in Two-Clock Timed Automata Is PSPACE-Complete. In Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming*, pages 212–223, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

17 John Fearnley and Marcin Jurdzinski. Reachability in two-clock timed automata is pspace-complete. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming – 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 212–223. Springer, 2013. doi:10.1007/978-3-642-39212-2_21.

18 Diego Figueira, Ranko Lazic, Jérôme Leroux, Filip Mazowiecki, and Grégoire Sutre. Polynomial-space completeness of reachability for succinct branching VASS in dimension one. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 119:1–119:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.119.

19 Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001. doi:10.1016/S0304-3975(00)00102-X.

20 Estíbaliz Fraca and Serge Haddad. Complexity analysis of continuous Petri nets. *Fundamenta Informaticae*, 137(1):1–28, 2015. doi:10.3233/FI-2015-1168.

21 Moses Ganardi, Rupak Majumdar, Andreas Pavlogiannis, Lia Schütze, and Georg Zetzsche. Reachability in bidirected pushdown VASS. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 124:1–124:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.124.

22 Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. doi:10.1145/146637.146681.

23 Stefan Göller, Christoph Haase, Ranko Lazic, and Patrick Totzke. A polynomial-time algorithm for reachability in branching VASS in dimension one. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 105:1–105:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.105.

24 Eitan M. Gurari and Oscar H. Ibarra. The complexity of decision problems for finite-turn multicounter machines. *J. Comput. Syst. Sci.*, 22(2):220–229, 1981. doi:10.1016/0022-0000(81)90028-3.

25 Christoph Haase and Simon Halfon. Integer vector addition systems with states. In Joël Ouaknine, Igor Potapov, and James Worrell, editors, *Reachability Problems – 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*, volume 8762 of *Lecture Notes in Computer Science*, pages 112–124. Springer, 2014. doi:10.1007/978-3-319-11439-2_9.

26 Christoph Haase, Stephan Kreutzer, Joël Ouaknine, and James Worrell. Reachability in succinct and parametric one-counter automata. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 – Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, volume 5710 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2009. doi:10.1007/978-3-642-04081-8_25.

27 John Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.

28 Eryk Kopczynski. Complexity of problems of commutative grammars. *Log. Methods Comput. Sci.*, 11(1), 2015. doi:10.2168/LMCS-11(1:9)2015.

29 Jérôme Leroux. The reachability problem for petri nets is not primitive recursive. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1241–1252. IEEE, 2021. doi:10.1109/FOCS52979.2021.00121.

30 Jérôme Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785796.

31 Jérôme Leroux and Grégoire Sutre. Reachability in Two-Dimensional Vector Addition Systems with States: One Test Is for Free. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2020.37.

32 Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. On the coverability problem for pushdown vector addition systems in one dimension. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming – 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2015. doi:10.1007/978-3-662-47666-6_26.

33 Richard Lipton. The reachability problem is exponential-space hard. *Yale University, Department of Computer Science, Report*, 62, 1976.

34 Filip Mazowiecki, Henry Sinclair-Banks, and Karol Węgrzycki. Coverability in 2-vass with one unary counter is in np. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures*, pages 196–217, Cham, 2023. Springer Nature Switzerland.

35 Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.

36 Louis E Rosier and Hsu-Chun Yen. A multiparameter analysis of the boundedness problem for vector addition systems. *Journal of Computer and System Sciences*, 32(1):105–135, 1986.

37 Sylvain Schmitz. The complexity of reachability in vector addition systems. *ACM SIGLOG News*, 3(1):4–21, 2016. doi:10.1145/2893582.2893585.

38 Leslie G. Valiant and Mike Paterson. Deterministic one-counter automata. *J. Comput. Syst. Sci.*, 10(3):340–350, 1975. doi:10.1016/S0022-0000(75)80005-5.

39 Wil M. P. van der Aalst. Verification of workflow nets. In *Proc. 18th International Conference on Application and Theory of Petri Nets (ICATPN)*, volume 1248, pages 407–426, 1997. doi:10.1007/3-540-63139-9_48.

Subtyping Context-Free Session Types

Gil Silva
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Andreia Mordido
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal

Abstract

Context-free session types describe structured patterns of communication on heterogeneously typed channels, allowing the specification of protocols unconstrained by tail recursion. The enhanced expressive power provided by non-regular recursion comes, however, at the cost of the decidability of subtyping, even if equivalence is still decidable. We present an approach to subtyping context-free session types based on a novel kind of observational preorder we call $\mathcal{XYZW}$-simulation, which generalizes $\mathcal{XY}$-simulation (also known as covariant-contravariant simulation) and therefore also bisimulation and plain simulation. We further propose a subtyping algorithm that we prove to be sound, and present an empirical evaluation in the context of a compiler for a programming language. Due to the general nature of the simulation relation upon which it is built, this algorithm may also find applications in other domains.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Concurrency; Software and its engineering → Concurrent programming structures

Keywords and phrases Session types, Subtyping, Simulation, Simple grammars, Non-regular recursion

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.11

Related Version *Technical Report*: https://arxiv.org/abs/2307.05661 [49]

Funding Support for this research was provided by the Fundação para a Ciência e a Tecnologia through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, and by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

Acknowledgements We thank Luca Padovani, Diana Costa, Diogo Poças and the anonymous reviewers for their insightful comments.

1 Introduction

Session types, introduced by Honda et al. [31, 32, 50], enhance traditional type systems with the ability to specify and enforce structured communication protocols on bidirectional, heterogeneously typed channels. Typically, these specifications include the type, direction (input or output) and order of the messages, as well as branching points where one participant can choose how to proceed and the other must follow.

Traditional session types are bound by tail recursion and therefore restricted to the specification of protocols described by regular languages. This excludes many protocols of practical interest, with the quintessential example being the serialization of tree-structured data on a single channel. Context-free session types, proposed by Thiemann and Vasconcelos [51], liberate types from tail recursion by introducing a sequential composition operator (_;_) with a monoidal structure and a left and right identity in type Skip, representing no action. As their name hints, context-free session types can specify protocols corresponding to (simple deterministic) context-free languages and are thus considerably more expressive than their regular counterparts.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 11; pp. 11:1–11:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

What does it mean for a context-free session type to be a subtype of another? Our answer follows Gay and Hole's seminal work on subtyping for regular session types [25], and Liskov's *principle of safe substitution* [39]: S is a subtype of R if channels governed by type S can take the place of channels governed by type R in whatever context, without violating the guarantees offered by a type system (e.g. progress, deadlock freedom, session fidelity, etc.).

More concretely, subtyping allows increased flexibility in the interactions between participants, namely on the type of the messages (a feature inherited from the subtyped π-calculus [46]) and on the choices available at branching points [25], allowing a channel to be governed by a simpler session type if its context so requires. A practical benefit of this flexibility is that it promotes *modular development*: the behaviour of one participant may be refined, while the behaviour of the other is kept intact.

▶ **Example 1.** Consider the following context-free session types for serializing binary trees.

$$\begin{aligned}
\mathsf{STree} &= \mu s.\oplus\{\mathsf{Nil}\colon \mathsf{Skip}, \mathsf{Node}\colon s;!\mathsf{Int};s\} & \mathsf{SEmpty} &= \oplus\{\mathsf{Nil}\colon \mathsf{Skip}\}\\
\mathsf{DTree} &= \mu s.\&\{\mathsf{Nil}\colon \mathsf{Skip}, \mathsf{Node}\colon s;?\mathsf{Int};s\} & \mathsf{SFullTree0} &= \oplus\{\mathsf{Node}\colon \mathsf{SEmpty};!\mathsf{Int};\mathsf{SEmpty}\}\\
& & \mathsf{SFullTree1} &= \oplus\{\mathsf{Node}\colon \mathsf{SFullTree0};!\mathsf{Int};\mathsf{SFullTree0}\}
\end{aligned}$$

The recursive STree and DTree types specify, respectively, the serialization and deserialization of a possibly infinite arbitrary tree, while the remaining non-recursive types specify the serialization of finite trees of particular configurations. The benefit of subtyping is that it makes the particular types SEmpty, SFullTree0 and SFullTree1 compatible with the general DTree type. Observe that its dual, STree, may safely take the place of any type in the right column. Consider now a function f that generates full trees of height 1 and serializes them on a given channel end. Assigning it type STree → Unit would not statically ensure that the fullness and height of the tree are as specified. Type SFullTree1 → Unit would do so, and subtyping would still allow the function to use an STree channel (i.e., communicate with someone expecting an arbitrary DTree tree).

Expressive power usually comes at the cost of decidability. While subtyping for regular session types has been formalized, shown decidable and given an algorithm by Gay and Hole [25], subtyping in the context-free setting has been proven undecidable by Padovani [43]. The proof is given by a reduction from the inclusion problem for simple languages, shown undecidable by Friedman [21]. Remarkably, the equivalence problem for simple languages is known to be decidable, as is the type equivalence of context-free session types [36, 51].

Subtyping context-free session types has until now been considered only in a limited form, where message types must be syntactically equal [43]. Consequently, the interesting co/contravariant properties of input/output types have been left unexplored. In this paper, we propose a more expressive subtyping relation, where the types of messages may vary co/contravariantly, according to the classical subtyping notion of Gay and Hole. To handle the contravariance of output types, we introduce a novel notion of observational preorder, which we call *$\mathcal{XYZW}$-simulation* (by analogy with *$\mathcal{XY}$-simulation* [1]).

While initially formulated in the context of the π-calculus, considerable work has been done to integrate session types in more standard settings, such as functional languages based on the polymorphic λ-calculus with linear types [2, 16, 47]. In this scenario, functional types and session types are not orthogonal: sessions may carry functions, and functions may act on sessions. With this in mind, we promote our theory to a linear functional setting, thereby showing how subtyping for records, variants and (linear and unrestricted [22]) functions, usually introduced by inference rules, can be seamlessly integrated with simulation-based subtyping for context-free session types.

Functional and higher-order context-free session types

$$\begin{aligned} T,U,V,W &::= \mathsf{Unit} \mid T \xrightarrow{m} U \mid (\!|\ell\colon T|\!)_{\ell\in L} \mid S \mid t \mid \mu t.T \\ S,R &::= \sharp T \mid \odot\{\ell\colon T\}_{\ell\in L} \mid \mathsf{Skip} \mid \mathsf{End} \mid S;R \mid s \mid \mu s.S \end{aligned}$$

Multiplicities, records/variants, polarities and views

$$m,n ::= \mathsf{1} \mid * \qquad (\!|\cdot|\!) ::= \{\cdot\} \mid \langle\cdot\rangle \qquad \sharp ::= ? \mid ! \qquad \odot ::= \oplus \mid \&$$

Figure 1 Syntax of types.

Finally, we present a sound algorithm for the novel notion of subtyping, based on the type equivalence algorithm of Almeida et al. [4]. This algorithm works by first encoding the types as words in a simple grammar [36] and then deciding their $\mathcal{XYZW}$-similarity. Being grammar-based and, at its core, agnostic to types, our algorithm may also find applications for other objects with similar non-regular and contravariant properties.

Contributions. We address the subtyping problem for context-free session types, proposing:

- A syntactic definition of subtyping for context-free session types;
- A novel kind of behavioural preorder called $\mathcal{XYZW}$-simulation, and, based on it, a semantic definition of subtyping that coincides with the syntactic one;
- A sound subtyping algorithm based on the $\mathcal{XYZW}$-similarity of simple grammars;
- An empirical evaluation of the performance of the algorithm, and a comparison with an existing type equivalence algorithm.

Overview. The rest of this paper is organized as follows: in Section 2 we introduce types, type formation and syntactic subtyping; in Section 3 we present a notion of semantic subtyping, to be used as a stepping stone to develop our subtyping algorithm; in Section 4 we present the algorithm and show it to be sound with respect to the semantic subtyping relation; in Section 5 we evaluate the performance of our implementation of the algorithm; in Section 6 we present related work; in Section 7 we conclude the paper and trace a path for the work to follow. The reader can find the rules for type formation and proofs for all results in the paper in a technical report on arXiv [49].

2 Types and syntactic subtyping

We base our contributions on a type language that includes both functional types and higher-order context-free session types (i.e., types that allow messages of arbitrary types). The language is shown in Figure 1. As customary in session types for functional languages [26], the language of types is given by two mutually recursive syntactic categories: one for functional types and another for session types. We assume two disjoint and denumerable sets of type references, with the first ranged over by t, u, v, w, the second by r, s and their union by x, y, z. We further assume a set of record, variant and choice labels, ranged over by j, k, ℓ.

The first three productions of the grammar for functional types introduce the Unit type, functions $T \xrightarrow{m} U$, records $\{\ell\colon T_\ell\}_{\ell\in L}$ and variants $\langle\ell\colon T_\ell\rangle_{\ell\in L}$ (which correspond to *datatypes* in ML-like languages). Our system exhibits linear characteristics: function types contain a multiplicity annotation m (also in Figure 1), meaning that they must be used exactly once if $m = 1$ or without restrictions if $m = *$ (such types can also be found, for instance, in Gay's proposal [26], in System F° [40] and in the FreeST language [2]). Their inclusion in our system is justified by the interesting subtyping properties they exhibit [22].

Session types $!T$ and $?T$ represent the sending and receiving, respectively, of a value of type T (an arbitrary type, making the system higher-order). Internal choice types $\oplus\{\ell\colon S_\ell\}_{\ell\in L}$ allow the selection of a label $k \in L$ and its continuation S_k, while external choice types $\&\{\ell\colon S_\ell\}_{\ell\in L}$ represent the branching on any label $k \in L$ and its continuation S_k. We stipulate that the set of labels for these types must be non empty. Type Skip represents no action, while type End indicates the closing of a channel, after which no more communication can take place. Type $R;S$ denotes the sequential composition of R and S, which is associative, right distributes over choices types, has (left and right) identity Skip and left-absorber End.

The final two productions in both functional and session grammars introduce self-references and the recursion operator. Their inclusion in the two grammars ensures we can have both recursive functional types and recursive session types while avoiding nonsensical types such as $\mu t.\mathsf{Unit} \overset{*}{\to} !\mathsf{Unit};t$ at the syntactical level (avoiding the need for a kinding system).

Still, we do not consider all types generated by these grammars to be *well-formed*. Consider session type $\mu r.r;!\mathsf{Unit}$. No matter how many times we unfold it, we cannot resolve its first communication action. The same could be said of $\mu r.\mathsf{Skip};r;!\mathsf{Unit}$. We must therefore ensure that any self-reference in a sequential composition is preceded by a type constructor representing some meaningful action, i.e., not equivalent to Skip. This is achieved by adapting the conventional notion of contractivity (no subterms of the form $\mu x.\mu x_1.\ \ldots\ \mu x_n.x$) [25] to account for Skip as the identity of sequential composition. This corresponds to the notion of *guardedness* in the theory of process algebra (e.g. [28, 42]).

In addition to contractivity, we must ensure that well-formed types contain no free references. The type formation judgement $\Delta \vdash T$, where Δ is a set of references, combines these requirements. The rules for the judgement can be found in the technical report [49].

We are now set to define our syntactic subtyping relation. We begin by surveying the features it should support:

Input and output subtyping. Input variance and output contravariance are the central features of subtyping for types that govern entities that can be written to or read from, such as channels and references [45]. They are therefore natural features of the subtyping relation for conventional session types as well [25]. Observe that $?\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\} \leq ?\{\mathsf{A}\colon \mathsf{Int}\}$ should be true, for the type of the received value, $\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$, safely substitutes the expected type, $\{\mathsf{A}\colon \mathsf{Int}\}$. Observe also that $!\{\mathsf{A}\colon \mathsf{Int}\} \leq !\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$ should be true, because the type of the value to be sent, $\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$, is a subtype of $\{\mathsf{A}\colon \mathsf{Int}\}$, the type of the messages the substitute channel is allowed to send.

Choice subtyping. If we understand external and internal choice types as, respectively, the input and output of a label, then their subtyping properties are easy to derive: external choices are covariant on their label set, internal choices are contravariant on their label set, and both are covariant on the continuation of the labels (this is known as *width subtyping*). Observe that $\&\{\mathsf{A}\colon ?\mathsf{Int}\} \leq \&\{\mathsf{A}\colon ?\mathsf{Int}, \mathsf{B}\colon !\mathsf{Bool}\}$ should be true, for every branch in the first type can be safely handled by matching on the second type. Likewise, $\oplus\{\mathsf{A}\colon ?\mathsf{Int}, \mathsf{B}\colon !\mathsf{Bool}\} \leq \oplus\{\mathsf{A}\colon ?\mathsf{Int}\}$ should be true, for every choice in the second type can be safely selected in the first.

Sequential composition. In the classical subtyping relation for regular session types, input and output types ($\sharp T.S$) can be characterized as covariant in their continuation. Although the same general intuition applies in the context-free setting, we cannot as easily characterize the variance of the sequential composition constructor ($S;R$) due to its monoidal, Syntactic subtyping (*coinductive*) $\boxed{T \le T}$

$$\textsc{S-Unit}\quad \mathsf{Unit} \le \mathsf{Unit} \qquad \textsc{S-Arrow}\quad \frac{U_1 \le T_1 \qquad T_2 \le U_2 \qquad m \sqsubseteq n}{T_1 \xrightarrow{m} T_2 \le U_1 \xrightarrow{n} U_2} \qquad \textsc{S-Rcd}\quad \frac{K \subseteq L \qquad T_j \le U_j \ (\forall j \in K)}{\{\ell\colon T_\ell\}_{\ell\in L} \le \{k\colon U_k\}_{k\in K}}$$

$$\textsc{S-Vrt}\quad \frac{L \subseteq K \qquad T_j \le U_j \ (\forall j \in L)}{\langle \ell\colon T_\ell\rangle_{\ell\in L} \le \langle k\colon U_k\rangle_{k\in K}} \qquad \textsc{S-RecL}\quad \frac{[\mu x.T/x]T \le U}{\mu x.T \le U} \qquad \textsc{S-RecR}\quad \frac{T \le [\mu x.U/x]U}{T \le \mu x.U} \qquad \textsc{S-In}\quad \frac{T \le U}{?T \le ?U}$$

$$\textsc{S-Out}\quad \frac{U \le T}{!T \le !U} \qquad \textsc{S-ExtChoice}\quad \frac{L \subseteq K \qquad S_j \le R_j \ (\forall j \in L)}{\&\{\ell\colon S_\ell\}_{\ell\in L} \le \&\{k\colon R_k\}_{k\in K}} \qquad \textsc{S-IntChoice}\quad \frac{K \subseteq L \qquad S_j \le R_j \ (\forall j \in K)}{\oplus\{\ell\colon S_\ell\}_{\ell\in L} \le \oplus\{k\colon R_k\}_{k\in K}}$$

$$\textsc{S-Skip}\quad \mathsf{Skip} \le \mathsf{Skip} \qquad \textsc{S-End}\quad \mathsf{End} \le \mathsf{End} \qquad \textsc{S-InSeq1L}\quad \frac{T \le U \qquad S \le \mathsf{Skip}}{?T;S \le ?U} \qquad \textsc{S-InSeq1R}\quad \frac{T \le U \qquad S \le \mathsf{Skip}}{?T \le ?U;S}$$

$$\textsc{S-InSeq2}\quad \frac{T \le U \qquad S \le R}{?T;S \le ?U;R} \qquad \textsc{S-OutSeq1L}\quad \frac{U \le T \qquad S \le \mathsf{Skip}}{!T;S \le !U} \qquad \textsc{S-OutSeq1R}\quad \frac{U \le T \qquad S \le \mathsf{Skip}}{!T \le !U;S} \qquad \textsc{S-OutSeq2}\quad \frac{U \le T \qquad S \le R}{!T;S \le !U;R}$$

$$\textsc{S-ChoiceSeqL}\quad \frac{\odot\{\ell\colon S_\ell;S\}_{\ell\in L} \le R}{\odot\{\ell\colon S_\ell\}_{\ell\in L};S \le R} \qquad \textsc{S-ChoiceSeqR}\quad \frac{S \le \odot\{\ell\colon R_\ell;R\}_{\ell\in L}}{S \le \odot\{\ell\colon R_\ell\}_{\ell\in L};R} \qquad \textsc{S-SkipSeqL}\quad \frac{S \le R}{\mathsf{Skip};S \le R} \qquad \textsc{S-SkipSeqR}\quad \frac{S \le R}{S \le \mathsf{Skip};R}$$

$$\textsc{S-EndSeq1L}\quad \mathsf{End};S \le \mathsf{End} \qquad \textsc{S-EndSeq1R}\quad \mathsf{End} \le \mathsf{End};R \qquad \textsc{S-EndSeq2}\quad \mathsf{End};S \le \mathsf{End};R \qquad \textsc{S-SeqSeqL}\quad \frac{S_1;(S_2;S_3) \le R}{(S_1;S_2);S_3 \le R} \qquad \textsc{S-SeqSeqR}\quad \frac{S \le R_1;(R_2;R_3)}{S \le (R_1;R_2);R_3}$$

$$\textsc{S-RecSeqL}\quad \frac{([\mu s.S_1/s]S_1);S_2 \le R}{(\mu s.S_1);S_2 \le R} \qquad \textsc{S-RecSeqR}\quad \frac{S \le ([\mu s.R_1/s]R_1);R_2}{S \le (\mu s.R_1);R_2}$$

Preorder on multiplicities $\boxed{m \sqsubseteq m}$

$$m \sqsubseteq m \qquad * \sqsubseteq 1$$

Figure 2 Syntactic subtyping.

distributive and absorbing properties. For instance, consider types $S_1;S_2$ and $R_1;R_2$, with $S_1 = !\mathsf{Int};!\mathsf{Bool}$, $S_2 = ?\mathsf{Int}$, $R_1 = !\mathsf{Int}$ and $R_2 = !\mathsf{Bool};?\mathsf{Int}$. Although it should be true that $S_1;S_2 \le R_1;R_2$, we can have neither $S_1 \le R_1$ nor $S_2 \le R_2$.

Functional subtyping. The subtyping properties of function, record and variant types are well known, and we refer the readers to Pierce's book for the reasoning behind them [45]. Succinctly, the function type constructor is contravariant on the domain and covariant on the range, and the variant and record constructors are both covariant on the type of the fields, but respectively covariant and contravariant on their label sets.

Multiplicity subtyping. Using an unrestricted ($*$) resource where a linear (1) one is expected does not compromise safety, provided that, multiplicities aside, the type of the former may safely substitute the type of the latter. We can express this relationship between multiplicities through a preorder captured by inequality $* \sqsubseteq 1$. In our system, function types may be either linear or unrestricted. Thus, type $T_1 \overset{m}{\to} T_2$ can be considered a subtype of $U_1 \overset{n}{\to} U_2$ if U_1 and T_2 are subtypes, respectively, of T_1 and U_2 and if $m \sqsubseteq n$ (thus we can characterize the function type constructor as covariant on its multiplicity).

The rules for our syntactic subtyping relation, interpreted coinductively, are shown in Figure 2. Rules S-Unit, S-Arrow, S-Rcd, S-Vrt, S-RecL and S-RecR establish the classical subtyping properties associated with both functional and equi-recursive types, with S-Arrow additionally encoding subtyping between linear and unrestricted functions, relying on a preorder on multiplicities also defined in Figure 2. Rules S-End, S-In, S-Out, S-ExtChoice and S-IntChoice bring to the context-free setting the classical subtyping properties expected from session types, as put forth by Gay and Hole [25].

The remaining rules account for sequential composition, which distributes over choice and exhibits a monoidal structure with its neutral element in Skip and left-absorbing element in End. We include, for each session type constructor S, a left rule (denoted by suffix L) of the form $S;R \leq S'$ and a right rule (denoted by suffix R) of the form $S' \leq S;R$. An additional rule is necessary for each constructor over which sequential composition does not distribute, associate or neutralize (S-InSeq2, S-OutSeq2 and S-EndSeq2). Since we are using a coinductive proof scheme, we include rules to "move" sequential composition down the syntax. Thus, given a type $S;R$, we inspect S to decide which rule to apply next.

▶ **Theorem 2.** *The syntactic subtyping relation $\leq$ is a preorder on types.*

▶ **Example 3.** Let us briefly return to Example 1. It is now easy to see that STree $\leq$ SFullTree1: we unfold the left-hand side and apply rule S-IntChoice. Then we apply the distributivity rules as necessary until reaching an internal choice with no continuation, at which point we can apply S-IntChoice again, or until reaching a type with !Int at the head, at which point we apply S-InSeq2. We repeat this process until reaching STree $\leq$ SFullTree0, and proceed similarly until reaching STree $\leq$ SEmpty, which follows from S-IntChoice and S-Skip.

Despite clearly conveying the intended meaning of the subtyping relation, the rules suggest no obvious algorithmic intepretation: on the one hand, the presence of bare metavariables makes the system not syntax-directed; on the other hand, rules S-RecL, S-RecSeqL and their right counterparts lead to infinite derivations which are not solvable by a conventional fixed-point construction [25, 45]. In the next section we develop an alternative, semantic approach to subtyping, which we use as a stepping stone to develop our subtyping algorithm.

3 Semantic subtyping

Semantic equivalence for context-free session types is usually based on *observational equivalence* or *bisimilarity*, meaning that two session types are considered equivalent if they exhibit exactly the same communication behaviour [51]. An analogous notion of *semantic subtyping* should therefore rely on an *observational preorder*. In this section we develop such a preorder.

We define the behaviour of types via a labelled transition system (LTS) by establishing relation $T \xrightarrow{a} U$ ("type T transitions by action a to type U"). We follow Costa et al. [16] in attributing behaviour to functional types, allowing them to be encompassed in our observational preorder. The rules defining the transition relation, as well as the grammar that generates all possible transition actions, are shown in Figure 3.

Labelled transition system $\boxed{T \xrightarrow{a} T}$

$$\textsc{L-Unit}\quad \mathsf{Unit} \xrightarrow{\mathsf{Unit}} \mathsf{Skip} \qquad \textsc{L-ArrowDom}\quad (T \overset{m}{\to} U) \xrightarrow{\to\mathsf{d}} T \qquad \textsc{L-ArrowRng}\quad (T \overset{m}{\to} U) \xrightarrow{\to\mathsf{r}} U \qquad \textsc{L-LinArrow}\quad (T \overset{1}{\to} U) \xrightarrow{\to\mathsf{1}} \mathsf{Skip}$$

$$\textsc{L-RcdVrtField}\quad \frac{k \in L}{\llparenthesis \ell\colon T_\ell \rrparenthesis_{\ell\in L} \xrightarrow{\llparenthesis\rrparenthesis_k} T_k} \qquad \textsc{L-RcdVrt}\quad \llparenthesis \ell\colon T_\ell \rrparenthesis_{\ell\in L} \xrightarrow{\llparenthesis\rrparenthesis} \mathsf{Skip} \qquad \textsc{L-Rec}\quad \frac{[\mu x.T/x]T \xrightarrow{a} U}{\mu x.T \xrightarrow{a} U} \qquad \textsc{L-Msg1}\quad \sharp T \xrightarrow{\sharp\mathsf{p}} T \qquad \textsc{L-Msg2}\quad \sharp T \xrightarrow{\sharp\mathsf{c}} \mathsf{Skip}$$

$$\textsc{L-Choice}\quad \odot\{\ell\colon S_\ell\}_{\ell\in L} \xrightarrow{\odot} \mathsf{Skip} \qquad \textsc{L-ChoiceField}\quad \frac{k \in L}{\odot\{\ell\colon S_\ell\}_{\ell\in L} \xrightarrow{\odot_\mathsf{k}} S_k} \qquad \textsc{L-End}\quad \mathsf{End} \xrightarrow{\mathsf{End}} \mathsf{Skip} \qquad \textsc{L-MsgSeq1}\quad \sharp T;S \xrightarrow{\sharp\mathsf{p}} T \qquad \textsc{L-MsgSeq2}\quad \sharp T;S \xrightarrow{\sharp\mathsf{c}} S$$

$$\textsc{L-ChoiceSeq}\quad \odot\{\ell\colon S_\ell\}_{\ell\in L};R \xrightarrow{\odot} \mathsf{Skip} \qquad \textsc{L-SkipSeq}\quad \frac{S \xrightarrow{a} T}{\mathsf{Skip};\, S \xrightarrow{a} T} \qquad \textsc{L-EndSeq}\quad \mathsf{End};S \xrightarrow{\mathsf{End}} \mathsf{Skip} \qquad \textsc{L-SeqSeq}\quad \frac{S_1;(S_2;S_3) \xrightarrow{a} T}{(S_1;S_2);S_3 \xrightarrow{a} T}$$

$$\textsc{L-ChoiceFieldSeq}\quad \frac{k \in L}{\odot\{\ell\colon S_\ell\}_{\ell\in L};R \xrightarrow{\odot_\mathsf{k}} S_k;R} \qquad \textsc{L-RecSeq}\quad \frac{([\mu s.S/s]S);R \xrightarrow{a} T}{(\mu s.S);R \xrightarrow{a} T} \qquad \text{(no rule for } \mathsf{Skip}\text{)}$$

Actions

$$a ::= \mathsf{Unit} \mid \to\mathsf{d} \mid \to\mathsf{r} \mid \to\mathsf{1} \mid \mathsf{End} \mid \llparenthesis\rrparenthesis_\ell \mid \llparenthesis\rrparenthesis \mid \sharp\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \odot_\ell$$

Figure 3 Labelled transition system. Letters d, r, p, c in labels stand for “domain”, “range”, “payload” and “continuation”.

In general, each functional type constructor generates a transition for each of its fields (Unit and End, which have none, transition to Skip). Linear functions exhibit an additional transition to represent their restricted use (L-LinArrow), and records/variants include a default transition that is independent of their fields (L-RcdVrt). The behaviour of session types is more complex, since it must account for their algebraic properties. Message types exhibit a transition for their payload (L-Msg1, L-MsgSeq1) and another for their continuation, which is Skip by omission (L-Msg2, L-MsgSeq2). Choices behave much like records/variants when alone, but are subject to distributivity when composed (L-ChoiceFieldSeq). Type End, which absorbs its continuation, transitions to Skip (L-End, L-EndSeq). Rules L-SeqSeq, L-SkipSeq account for associativity and identity, and rules L-Rec and L-RecSeq dictate that recursive types behave just like their unfoldings. Notice that Skip has no transitions.

With the behaviour of types established, we now look for an appropriate notion of observational preorder. Several such notions have been studied in the literature. *Similarity*, defined as follows, is arguably the simplest of them [41, 44].

▶ **Definition 4.** *A type relation $\mathcal{R}$ is said to be a simulation if, whenever $T\mathcal{R}U$, for all a and T' with $T \xrightarrow{a} T'$ there is U' such that $U \xrightarrow{a} U'$ and $T'\mathcal{R}U'$*

Similarity, written $\preceq$, is the union of all simulation relations. We say that a type U simulates type T if $T \preceq U$.

Unfortunately, plain similarity is of no use to us. A small example shows why: type $\oplus\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon \mathsf{End}\}$ both simulates and is a subtype of $\oplus\{\mathsf{A}\colon \mathsf{End}\}$, while type $\&\{\mathsf{A}\colon \mathsf{End}\}$ does not simulate yet is a subtype of $\&\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon \mathsf{End}\}$. Reversing the direction of the simulation would be of no avail either, as it would leave us with the reverse problem.

It is apparent that a more refined notion of simulation is necessary, where the direction of the implication depends on the transition labels. Aarts and Vaandrager provide just such a notion in the form of *$\mathcal{XY}$-simulation* [1], a simulation relation parameterized by two subsets of actions, $\mathcal{X}$ and $\mathcal{Y}$, such that actions in $\mathcal{X}$ are simulated from left to right and those in $\mathcal{Y}$ are simulated from right to left, selectively combining the requirements of simulation and reverse simulation.

▶ **Definition 5.** *Let $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{A}$. A type relation $\mathcal{R}$ is said to be an $\mathcal{XY}$-simulation if, whenever $T\mathcal{R}U$, we have:*

1. *for each $a \in \mathcal{X}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $T'\mathcal{R}U'$;*
2. *for each $a \in \mathcal{Y}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $T'\mathcal{R}U'$.*

$\mathcal{XY}$-similarity, written $\preceq^{\mathcal{XY}}$, is the union of all $\mathcal{XY}$-simulation relations. We say that a type T is $\mathcal{XY}$-similar to type U if $T \preceq^{\mathcal{XY}} U$.

Similar or equivalent notions have appeared throughout the literature: *modal refinement* [38], *alternating simulation* [7] and, perhaps more appropriately named (for our purposes), *covariant-contravariant simulation* [20]. Padovani's original subtyping relation for context-free session types [43] can also be understood as a refined form of $\mathcal{XY}$-simulation.

We can tentatively define a semantic subtyping relation $\lesssim'$ as $\mathcal{XY}$-similarity, where $\mathcal{X}$ and $\mathcal{Y}$ are the label sets generated by the following grammars for $a_{\mathcal{X}}$ and $a_{\mathcal{Y}}$, respectively.

$$a_{\mathcal{X}} ::= a_{\mathcal{XY}} \mid \langle\rangle_\ell \mid \&_\ell \qquad\qquad a_{\mathcal{XY}} ::= \mathsf{Unit} \mid {\rightarrow}\mathsf{d} \mid {\rightarrow}\mathsf{r} \mid \llparenthesis\rrparenthesis \mid \sharp\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \mathsf{End}$$
$$a_{\mathcal{Y}} ::= a_{\mathcal{XY}} \mid {\rightarrow}\mathsf{1} \mid \{\}_\ell \mid \oplus_\ell$$

This would indeed give us the desired result for our previous example, but we still cannot account for the contravariance of output and function types: we want $T = !\{\mathsf{A}\colon \mathsf{Int}\}$ to be a subtype of $U = !\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$, yet $T \lesssim' U$ does not hold (in fact, we have $U \lesssim' T$, a clear violation of run-time safety). The same could be said for types $\{\mathsf{A}\colon \mathsf{Int}\} \overset{*}{\rightarrow} \mathsf{Int}$ and $\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\} \overset{*}{\rightarrow} \mathsf{Int}$. In short, our simulation needs the $!\mathsf{p}$ and ${\rightarrow}\mathsf{d}$-derivatives to be related in the direction opposite to that of the initial types. Thus we need to selectively apply a strong form of *contrasimulation* as well [48, 52] (the original notion is defined with weak transitions, a sort of transition we do not address).

To allow this, we generalize the definition of $\mathcal{XY}$-simulation by parameterizing it on two further subsets of actions and including two more clauses where the direction of the relation between the derivatives is reversed. By analogy with $\mathcal{XY}$-simulation, we call the resulting notion $\mathcal{XYZW}$-simulation.

▶ **Definition 6.** *Let $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W} \subseteq \mathcal{A}$. A type relation $\mathcal{R}$ is a $\mathcal{XYZW}$-simulation if, whenever $T\mathcal{R}U$, we have:*

1. *for each $a \in \mathcal{X}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $T'\mathcal{R}U'$;*
2. *for each $a \in \mathcal{Y}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $T'\mathcal{R}U'$;*
3. *for each $a \in \mathcal{Z}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $U'\mathcal{R}T'$;*
4. *for each $a \in \mathcal{W}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $U'\mathcal{R}T'$.*

$\mathcal{XYZW}$-similarity, written $\preceq^{\mathcal{XYZW}}$, is the union of all $\mathcal{XYZW}$-simulation relations. We say that a type T is $\mathcal{XYZW}$-similar to type U if $T \preceq^{\mathcal{XYZW}} U$.

$\mathcal{XYZW}$-simulation generalizes several existing observational relations: $\mathcal{XY}$-simulation can be defined as an $\mathcal{XY}\emptyset\emptyset$-simulation, bisimulation as $\mathcal{AA}\emptyset\emptyset$-simulation (alternatively, $\emptyset\emptyset\mathcal{AA}$-simulation or $\mathcal{AAAA}$-simulation), and plain simulation as $\mathcal{A}\emptyset\emptyset\emptyset$-simulation.

▶ **Theorem 7.** *For any $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W}$, $\preceq^{\mathcal{XYZW}}$ is a preorder relation on types.*

Equipped with the notion of $\mathcal{XYZW}$-similarity, we are ready to define the semantic subtyping relation for functional and higher-order context-free session types as follows.

▶ **Definition 8.** *The semantic subtyping relation for functional and higher-order context-free session types $\lesssim$ is defined by $T \lesssim U$ when $T \preceq^{\mathcal{XYZW}} U$ such that $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$ and $\mathcal{W}$ are defined as the label sets generated by the following grammars for $a_\mathcal{X}$, $a_\mathcal{Y}$, $a_\mathcal{Z}$ and $a_\mathcal{W}$, respectively.*

$$\begin{array}{ll} a_\mathcal{X} ::= a_\mathcal{XY} \mid \rightarrow\mathsf{1} \mid \langle\rangle_\ell \mid \&_\ell & a_\mathcal{Z}, a_\mathcal{W} ::= !\mathsf{p} \mid \rightarrow\mathsf{d} \\ a_\mathcal{Y} ::= a_\mathcal{XY} \mid \{\}_\ell \mid \oplus_\ell & a_\mathcal{XY} ::= \mathsf{Unit} \mid \rightarrow\mathsf{r} \mid \llparenthesis\rrparenthesis \mid ?\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \mathsf{End} \end{array}$$

Notice the correspondence between the placement of the labels and the variance of their respective type constructors. Labels arising from covariant positions of the arrow and input type constructors are placed in both the $\mathcal{X}$ and $\mathcal{Y}$ sets, while those arising from the contravariant positions of the arrow and output type constructors are placed in both the $\mathcal{Z}$ and $\mathcal{W}$ sets. Labels arising from the fields of constructors exhibiting width subtyping are placed in a single set, depending on the variance of the constructor on the label set: $\mathcal{X}$ for covariance (external choice and variant constructors), $\mathcal{Y}$ for contravariance (internal choice and record constructors). The function type constructor is covariant on its multiplicity, thus the linear arrow label is placed in $\mathcal{X}$. Finally, default record/variant/choice labels and those arising from nullary constructors are placed in $\mathcal{X}$ and $\mathcal{Y}$, but they could alternatively be placed in $\mathcal{Z}$ and $\mathcal{W}$ or in all four sets (notice the parallel with bisimulation, that can be defined as $\mathcal{AA}\emptyset\emptyset$-simulation, $\emptyset\emptyset\mathcal{AA}$-simulation, or $\mathcal{AAAA}$-simulation).

▶ **Example 9.** Let us go back once again to our tree serialization example from Section 1. Here it is also easy to see that $\mathsf{STree} \lesssim \mathsf{SFullTree1}$. Observe that, on the side of STree, transitions by $\oplus_\mathsf{Nil}$ and $\oplus_\mathsf{Node}$ always appear together, while on the side of $\mathsf{SFullTree1}$ types transition first by $\oplus_\mathsf{Node}$ and then by $\oplus_\mathsf{Nil}$. Since $\oplus_\mathsf{Nil}$ and $\oplus_\mathsf{Node}$ belong exclusively to $\mathcal{Y}$, STree is always able to match $\mathsf{SFullTree1}$ on these labels (as in all the others in $\mathcal{Y} \cup \mathcal{W}$, and *vice-versa* for $\mathcal{X} \cup \mathcal{Z}$).

▶ **Theorem 10** (Soundness and completeness for subtyping relations)**.** *Let $\vdash T$ and $\vdash U$. Then $T \leq U$ iff $T \lesssim U$.*

4 A subtyping algorithm

The notion of subtyping we have outlined is undecidable. This follows from the fact that our system, albeit different, contains all the features necessary to reconstruct Padovani's proof of undecidability [43]. Using just external choices, sequential composition, the Skip type and recursion, one is able to encode simple grammars [36] as context-free session types, in a way that language strings correspond to complete LTS traces of types. By exploiting the covariant width-subtyping in external choices, one can show that subtyping for these types corresponds to language inclusion, which is known to be undecidable for simple languages [21].

Despite the undecidability of our subtyping problem, we are still able to devise a sound (but necessarily incomplete) algorithm for it. In this section we present this algorithm, an adaptation of the equivalence algorithm of Almeida et al. [4]. At its core, it determines the $\mathcal{XYZW}$-similarity of simple grammars. Its application to context-free session types is facilitated by a translation function to properly encode types as grammars. The algorithm may likewise be adapted to other domains. Much like the original, our algorithm can be succinctly described in three distinct phases:

1. translate the given types to a simple grammar [36] and two starting words;
2. prune unreachable symbols from productions;
3. explore an expansion tree rooted at a node containing the initial words, alternating between expansion and simplification operations until either an empty node is found (decide **True**) or all nodes fail to expand (decide **False**).

Phase 1. The first phase consists of translating the two types to a grammar in *Greibach normal form* (GNF) [27], i.e., a grammar where all productions have the form $Y \rightarrow a\vec{Z}$, and two starting words $(\vec{X}, \vec{Y})$. A word is defined as a sequence of non-terminal symbols. We can check the $\mathcal{XYZW}$-similarity of words in GNF grammars because they naturally induce a labelled transition system, where states are words $\vec{X}$, actions are terminal symbols a and the transition relation is defined as $X\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Z}\vec{Y}$ when $X \rightarrow a\vec{Z} \in \mathcal{P}$. We denote the bisimilarity and $\mathcal{XYZW}$-similarity of grammars by, respectively, $\sim_{\mathcal{P}}$ and $\preceq_{\mathcal{P}}^{\mathcal{XYZW}}$, where $\mathcal{P}$ is the set of productions. We also let $\lesssim_{\mathcal{P}}$ denote grammar $\mathcal{XYZW}$-similarity with label sets as in Definition 8. The deterministic nature of context-free session types allows their corresponding grammars to be simple [36]: for each non-terminal Y and terminal symbol a, we have at most one production of the form $Y \rightarrow a\vec{Z}$.

The grammar translation procedure *grm* remains unchanged from the original equivalence algorithm [4], and for this reason we omit its details (which include generating productions for all μ-subterms in types). However, this procedure relies on two auxiliary definitions which must be adapted: the *unr* function (Definition 11), which normalizes the head of session types and unravels recursive types until reaching a type constructor, and the *word* procedure (Definition 12), which builds a word from a session type while updating a set $\mathcal{P}$ of productions.

▶ **Definition 11.** *The* unraveling *of a type T is defined by induction on the structure of T:*

$$\begin{aligned} unr(\mu x.T) &= unr([\mu x.T/x]T) & unr(\mathsf{Skip};S) &= unr(S) \\ unr(\mathsf{End};S) &= \mathsf{End} & unr((\mu s.S);R) &= unr(([\mu s.S/s]S);R) \\ unr(\odot\{\ell\colon S_\ell\}_{\ell\in L};R) &= \odot\{\ell\colon S_\ell;R\}_{\ell\in L} & unr((S_1;S_2);S_3) &= unr(S_1;(S_2;S_3)) \end{aligned}$$

and in all other cases by $unr(T) = T$.

▶ **Definition 12.** *The word corresponding to a well-formed type T, $word(T)$, is built by descending on the structure of T while updating a set $\mathcal{P}$ of productions:*

$$\begin{aligned} word(\mathsf{Unit}) &= Y,\ setting\ \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \rightarrow \mathsf{Unit}\} \\ word(U \overset{1}{\rightarrow} V) &= Y,\ setting\ \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \rightarrow {\rightarrow}\mathsf{d}\,word(U), Y \rightarrow {\rightarrow}\mathsf{r}\,word(V), Y \rightarrow {\rightarrow}1\} \\ word(U \overset{*}{\rightarrow} V) &= Y,\ setting\ \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \rightarrow {\rightarrow}\mathsf{d}\,word(U), Y \rightarrow {\rightarrow}\mathsf{r}\,word(V)\} \\ word(\llparenthesis \ell\colon T_\ell \rrparenthesis_{\ell\in L}) &= Y,\ setting\ \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \rightarrow \llparenthesis\rrparenthesis\bot\} \cup \{Y \rightarrow \llparenthesis\rrparenthesis_k\, word(T_k) \mid k \in L\} \\ word(\mathsf{Skip}) &= \varepsilon \\ word(\mathsf{End}) &= Y,\ setting\ \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \rightarrow \mathsf{End}\bot\} \\ word(\sharp U) &= Y\ ,\ setting\ \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \rightarrow \sharp\mathsf{p}\,word(U)\bot, Y \rightarrow \sharp\mathsf{c}\} \\ word(\odot\{\ell\colon S_\ell\}_{\ell\in L}) &= Y\ ,\ setting\ \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \rightarrow \odot\bot\} \cup \{Y \rightarrow \odot_{\mathsf{k}}\, word(S_k) \mid k \in L\} \\ word(S_1; S_2) &= word(S_1)\,word(S_2) \\ word(\mu x.U) &= X \end{aligned}$$

where, in each equation, Y is understood as a fresh non-terminal symbol, X as the non-terminal symbol corresponding to type reference x, and $\bot$ as a non-terminal symbol without productions.

▶ **Example 13.** Consider again the types for tree serialization in Section 1. Suppose we want to know whether $\mathsf{SFullTree0} \xrightarrow{*} \mathsf{Unit} \lesssim \mathsf{STree} \xrightarrow{1} \mathsf{Unit}$. We know that the grammar generated for these types is as follows, with X_0 and Y_0 as their starting words.

$$\begin{array}{lllll}
X_0 \to \rightarrowtail\mathsf{d}X_1 & X_2 \to \oplus_{\mathsf{Empty}} & X_4 \to \mathsf{Int} & Y_0 \to \rightarrowtail\mathsf{d}Y_1 & Y_1 \to \oplus\bot \\
X_0 \to \rightarrowtail\mathsf{r}X_5 & X_2 \to \oplus\bot & X_5 \to \mathsf{Unit} & Y_0 \to \rightarrowtail\mathsf{r}X_5 & Y_1 \to \oplus_{\mathsf{Empty}} \\
X_1 \to \oplus_{\mathsf{Node}}X_2X_3X_2 & X_3 \to !\mathsf{p}X_4\bot & & Y_0 \to \rightarrowtail 1 & Y_1 \to \oplus_{\mathsf{Node}}Y_1X_3Y_1 \\
X_1 \to \oplus\bot & X_3 \to !\mathsf{c} & & &
\end{array}$$

For the rest of this section let $\vdash T$, $\vdash U$, $(\vec{X}_T, \mathcal{P}') = \mathit{grm}(T, \emptyset)$ and $(\vec{X}_U, \mathcal{P}) = \mathit{grm}(U, \mathcal{P}')$.

▶ **Theorem 14** (Soundness for grammars). *If $\vec{X}_T \lesssim_{\mathcal{P}} \vec{X}_U$, then $T \lesssim U$.*

Phase 2. The grammars generated by procedure *grm* may contain unreachable words, which can be ignored by the algorithm. Intuitively, these words correspond to communication actions that cannot be fulfilled, such as subterm $?\mathsf{Bool}$ in type $(\mu s.!\mathsf{Int};s);?\mathsf{Bool}$. Formally, these words appear in productions following what are known as *unnormed words*.

▶ **Definition 15.** *Let $\vec{a}$ be a non-empty sequence of non-terminal symbols $a_1, \ldots, a_n$. Write $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$ when $\vec{Y} \xrightarrow{a_1}_{\mathcal{P}} \ldots \xrightarrow{a_n}_{\mathcal{P}} \vec{Z}$. We say that a word $\vec{Y}$ is normed if $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$ for some $\vec{a}$, and unnormed otherwise. If $\vec{Y}$ is normed and $\vec{a}$ is the shortest path such that $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$, then $\vec{a}$ is called the minimal path of $\vec{Y}$, and its length is the norm of $\vec{Y}$, denoted $|\vec{Y}|$.*

It is known that any unnormed word $\vec{Y}$ is bisimilar to its concatenation with any other word, i.e., if $\vec{Y}$ is unnormed, then $\vec{Y} \sim_{\mathcal{P}} \vec{Y}\vec{X}$. It is also easy to show that $\sim_{\mathcal{P}} \subseteq \lesssim_{\mathcal{P}}$, and hence that $\vec{Y} \lesssim_{\mathcal{P}} \vec{Y}\vec{X}$. In this case, $\vec{X}$ is said to be unreachable and can be safely removed from the grammar. We call the procedure of removing all unreachable symbols from a grammar *pruning*, and denote the pruned version of a grammar $\mathcal{P}$ by $\mathit{prune}(\mathcal{P})$.

▶ **Lemma 16** (Pruning preserves $\mathcal{XYZW}$-similarity). *$\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ iff $\vec{X} \preceq_{\mathit{prune}(\mathcal{P})}^{\mathcal{XYZW}} \vec{Y}$*

Phase 3. In its third and final phase, the algorithm explores an *expansion tree*, alternating between expansion and simplification steps. An expansion tree is a tree whose nodes are sets of pairs of words, whose root is the singleton set containing the pair of starting words under test, and where every child is an *expansion* of its parent. A branch is deemed *successful* if it is infinite or has an empty leaf, and deemed *unsuccessful* otherwise. The original definition of expansion ensures that the union of all nodes along a successful branch (without simplifications) constitutes a bisimulation [35]. We adapt this definition to ensure that such a union yields an $\mathcal{XYZW}$-simulation instead.

▶ **Definition 17.** *The* $\mathcal{XYZW}$-expansion *of a node N is defined as the minimal set N' such that, for every pair $(\vec{X}, \vec{Y})$ in N, it holds that:*

1. *if $\vec{X} \to a\vec{X}'$ and $a \in \mathcal{X}$ then $\vec{Y} \to a\vec{Y}'$ with $(\vec{X}', \vec{Y}') \in N'$*
2. *if $\vec{Y} \to a\vec{Y}'$ and $a \in \mathcal{Y}$ then $\vec{X} \to a\vec{X}'$ with $(\vec{X}', \vec{Y}') \in N'$*
3. *if $\vec{X} \to a\vec{X}'$ and $a \in \mathcal{Z}$ then $\vec{Y} \to a\vec{Y}'$ with $(\vec{Y}', \vec{X}') \in N'$*
4. *if $\vec{Y} \to a\vec{Y}'$ and $a \in \mathcal{W}$ then $\vec{X} \to a\vec{X}'$ with $(\vec{Y}', \vec{X}') \in N'$*

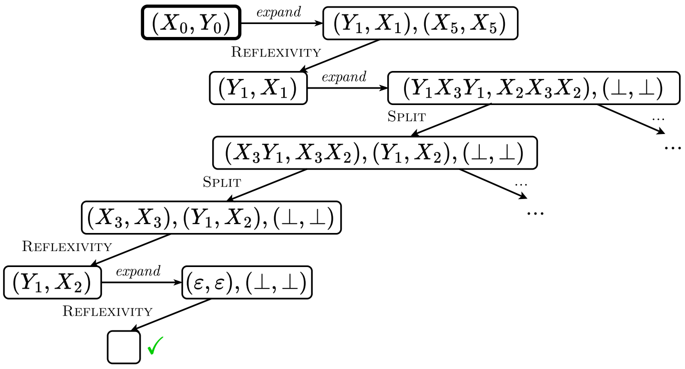

(X_0, Y_0)
expand
$(Y_1, X_1), (X_5, X_5)$
REFLEXIVITY
(Y_1, X_1)
expand
$(Y_1X_3Y_1, X_2X_3X_2), (\bot, \bot)$
SPLIT
...
...
$(X_3Y_1, X_3X_2), (Y_1, X_2), (\bot, \bot)$
SPLIT
...
...
$(X_3, X_3), (Y_1, X_2), (\bot, \bot)$
REFLEXIVITY
(Y_1, X_2)
expand
$(\varepsilon, \varepsilon), (\bot, \bot)$
REFLEXIVITY
✓

Figure 4 An $\mathcal{XYZW}$-expansion tree for Example 13, exhibiting a finite successful branch.

▶ **Lemma 18** (Safeness property for $\mathcal{XYZW}$-simulation). *Given a set of productions $\mathcal{P}$, $\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ iff the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a successful branch.*

The simplification stage consists of applying rules that safely modify the expansion tree during its construction, in an attempt to keep some branches finite. The rules are iteratively applied to each node until a fixed point is reached, at which point we can proceed with expansion. To each node N we apply three simplification rules, adapted from the equivalence algorithm [4]:

1. REFLEXIVITY: omit pairs of the form $(\vec{X}, \vec{X})$;
2. PREORDER: omit pairs belonging to the least preorder containing the ancestors of N;
3. SPLIT: if $(X_0\vec{X}, Y_0\vec{Y}) \in N$ and X_0 and Y_0 are normed, then:
   - Case $|X_0| \leq |Y_0|$: Let $\vec{a}$ be a minimal path for X_0 and $\vec{Z}$ the word such that $Y_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including pairs $(X_0\vec{Z}, Y_0)$ and $(\vec{X}, \vec{Z}\vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$;
   - Otherwise: Let $\vec{a}$ be a minimal path for Y_0 and $\vec{Z}$ the word such that $X_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including pairs $(X_0, Y_0\vec{Z})$ and $(\vec{Z}\vec{X}, \vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$.

When a node is simplified, we keep track of the original node in a sibling, thus ensuring that along the tree we keep an "expansion-only" branch.

The algorithm explores the tree by breadth-first search using a queue of node-ancestors pairs, thus avoiding getting stuck in infinite branches, and alternates between expansion and simplification steps until it terminates with **False** if all nodes fail to expand or with **True** if an empty node is reached. The following pseudo-code illustrates the procedure.

```
$subG(\vec{X}, \vec{Y}, \mathcal{P}) = explore(singletonQueue((\{(\vec{X}, \vec{Y})\}, \emptyset), \mathcal{P})$
  where $explore(q, \mathcal{P}) =$
    if $empty(q)$ then False % all nodes failed to expand
    else let $(n, a) = front(q)$ in
      if $empty(n)$ then True % empty node reached
      else if $hasExpansion(n, \mathcal{P})$ % then expand, simplify and recur
          then $explore(simplify(expand(n, \mathcal{P}), a \cup n, dequeue(q)), \mathcal{P})$
          else $explore(dequeue(q), \mathcal{P})$ % otherwise, discard node
```

▶ **Example 19.** The $\mathcal{XYZW}$-expansion tree for Example 13 is illustrated in Figure 4.

Finally, function *subT* puts all the pieces of the algorithm together:

$$\mathit{subT}(T, U) = \textbf{let } (\vec{X}, \mathcal{P}') = \mathit{grm}(T, \emptyset), (\vec{Y}, \mathcal{P}) = \mathit{grm}(U, \mathcal{P}') \textbf{ in } \mathit{subG}(\vec{X}, \vec{Y}, \mathit{prune}(\mathcal{P}))$$

It receives two well-formed types T and U, computes their grammar and respective starting words $\vec{X}$ and $\vec{Y}$, prunes the productions of the grammar and, lastly, uses function *subG* to determine whether $\vec{X} \lesssim_{\mathcal{P}} \vec{Y}$.

The following result shows that algorithm *subT* is sound with respect to semantic subtyping relation on functional and higher-order context-free session types.

▶ **Theorem 20** (Soundness). *If* $\mathit{subT}(T, U)$ *returns* **True**, *then* $T \lesssim U$.

5 Evaluation

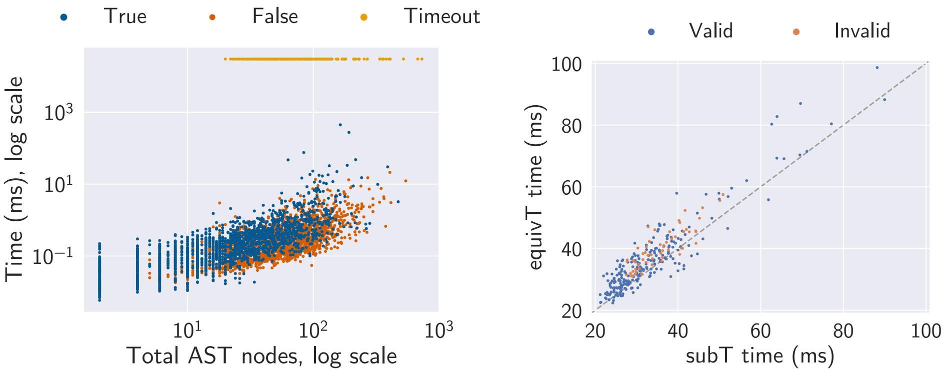
We have implemented our subtyping algorithm in Haskell and integrated it in the freely available compiler for FreeST, a statically typed functional programming language featuring message-passing channels governed by context-free session types [2, 3, 6]. The FreeST compiler features a running implementation of the type equivalence algorithm of Almeida et al. [4]. With our contributions, FreeST effectively gains support for subtyping at little to no cost in performance. In this section we present an empirical study to support this claim.

We employed three test suites to evaluate the performance of our algorithm: a suite of handwritten pairs of types, a suite of randomly generated pairs of types, and a suite of handwritten FreeST programs. We focus on the last two, since they allow a more robust and realistic analysis. All data was collected on a machine featuring an Intel Core i5-6300U at 2.4GHz with 16GB of RAM.

To build our randomly generated suite we employed a type generation module, implemented using the Quickcheck library [15] and following an algorithm induced from the properties of subtyping, much like the one induced by Almeida et al. [4] from the properties of bisimilarity. It includes generators for valid and invalid subtyping pairs. We conducted our evaluation by taking the running time of the algorithm on 2000 valid pairs and 2000 invalid pairs, ranging from 2 to 730 total AST nodes, with a timeout of 30s (ensuring it terminates with either **True**, **False** or **Unknown**). The results are plotted in Figure 5a. Despite the incompleteness of the algorithm, we encountered no false negatives, but obtained 188 timeouts. We found, as expected, that the running time increases considerably with the number of nodes. When a result was produced, valid pairs took generally longer.

Randomly generated types allow for a robust analysis, but they typically do not reflect the types encountered by a subtyping algorithm in its most obvious practical application, a compiler. For this reason, we turn our attention to our suite of FreeST programs, comprised of 286 valid and invalid programs collected throughout the development of the FreeST language. Programs range from small examples demonstrating particular features of the language to concurrent applications simulating, for example, an FTP server.

We began by integrating the algorithm in the FreeST compiler, placing next to every call to the original algorithm [4] (henceforth `equivT`) a call to `subT` on the same pairs of types. We then ran each program in our suite 10 times, collecting and averaging the accumulated running time of both algorithms on the same pairs of types. We then took the difference between the average accumulated running times of `subT` and `equivT`, obtaining an average difference of -3.85ms, with a standard deviation of 7.08ms, a minimum difference of -71.29ms and a maximum difference of 8.03ms (`subT` performed faster, on average). Figure 5b illustrates this comparison by plotting against each other the accumulated running times (for clarity, those in the 20-100ms range) of both algorithms during the typechecking phase of each.



(a) Performance on valid and invalid subtyping pairs

(b) Performance comparison against the original equivalence algorithm

Figure 5 Performance evaluation and comparison.

The data collected in this evaluation suggests that replacing the original equivalence algorithm [4] with the subtyping algorithm in the FreeST typechecker generally does not incur an overhead, while providing additional expressive power for programmers.

6 Related work

Session types emerged as a formalism to express communication protocols and statically verify their implementations [31, 32]. Initial formulations allowed only pairwise, tail-recursive protocols, earning such types the “binary” and “regular” epithets. Since then, considerable efforts have been made to extend the theory of session types beyond the binary and regular realms: multiparty session types allow sessions with multiple participants [33], while context-free session types [51] and nested session types [18] allow non-regular communication patterns. Our work is centered on context-free session types, which have seen considerable development since their introduction, most notably their integration in System F [2, 47], an higher-order formulation [16], as well as proposals for kind and type inference [5, 43].

Subtyping is a standard feature of many type systems, and the literature on the topic is vast [8, 10, 13, 14, 17, 19, 37]. Its conventional interpretation, based on the notion of substitutability, originates from the work of Liskov [39]. Multiple approaches to subtyping for regular session types have been proposed, and they can be classified according to the objects they consider substitutable: channels *versus* processes (the difference being most notable in the variance of type constructors). The earliest approach, subscribing to the substitutability of channels, is that of Gay and Hole [25]. It is also the one we follow. A later formulation, proposed by Carbone et al. [12], subscribes to the substitutability of processes. A survey of both interpretations is given by Gay [24]. The interaction between subtyping and polymorphism for regular session types, in the form of bounded quantification, has been investigated by Gay [23]. Horne and Padovani study subtyping under the linear logic interpretation of regular session types [34], showing that it preserves termination of processes.

Subtyping for session types has spread beyond the regular realm. Das et al. [18] introduce subtyping for nested session types, show the problem to be undecidable and present a sound but incomplete algorithm. In the context-free setting, the first and, to the best of our knowledge, only formulation before our work is that of Padovani [43]. It proposes a simulation-based subtyping relation, proves the undecidability of the subtyping problem and provides a sound but incomplete algorithm. This undecidability proof also applies to our system, as it possesses all the required elements: width-subtyping on choices, sequential composition and recursion. The subtyping relation proposed by Padovani contemplates neither input/output subtyping nor functional subtyping. Furthermore, its implementation relies on the subtyping features of OCaml, the implementation language. In contrast, we propose a more expressive relation, featuring input/output subtyping, as well as functional subtyping. Furthermore, we provide an also sound algorithm that is independent of the implementation language.

Our subtyping relation is based on a novel form of observational preorder, $\mathcal{XYZW}$-simulation. There is, as far as we know, no analogue in the literature. It is a generalization of $\mathcal{XY}$-simulation, introduced by Aarts and Vaandrager in the context of learning automata [1] but already known, under slightly different forms, as modal refinement [38], alternating simulation [7] and covariant-contravariant simulation [20]. The contravariance on the derivatives introduced by $\mathcal{XYZW}$-simulation is also prefigured in contrasimulation [48, 52], but the former uses strong transitions whereas the latter uses weak ones. There is a vast literature on other observational relations, to which Sangiorgi's book provides an overview [48].

Our algorithm decides the $\mathcal{XYZW}$-similarity of simple grammars [36]. It is an adaptation of the bisimilarity algorithm for simple grammars of Almeida et al. [4]. To our knowledge, these are the only running algorithms of their sort. Henry and Sénizergues [29] proposed an algorithm to decide the language equivalence problem on deterministic pushdown automata. On the related topic of basic process algebra (BPA), BPA processes have been shown to be equivalent to grammars in GNF [9], of which simple grammars are a particular case. This makes results and algorithms for BPA processes applicable to grammars in GNF, and *vice-versa*. A bisimilarity algorithm for general BPA processes, of doubly-exponential complexity, has been proposed by Burkart et al. [11], while an analogous polynomial-time algorithm for the special case of normed BPA processes has been proposed by Hirschfield et al. [30].

7 Conclusion and future work

We have proposed an intuitive notion of subtyping for context-free session types, based on a novel form of observational preorder, $\mathcal{XYZW}$-simulation. This preorder inverts the direction of the simulation in the derivatives covered by its $\mathcal{W}$ and $\mathcal{Z}$ parameters, allowing it to handle co/contravariant features of input/output types. We take advantage of the fact that $\mathcal{XYZW}$-simulation generalizes bisimulation to derive a sound subtyping algorithm from an existing type equivalence algorithm.

Despite its unavoidable incompleteness, stemming from the undecidability of our notion of subtyping, our algorithm has not yielded any false negatives. Thus, we conjecture that is partially correct: it may not halt, but, when it does, the answer is correct. We cannot, however, back this claim without a careful analysis of completeness and termination, which we leave for future work. We believe such an analysis will advance the understanding of the subtyping problem by clarifying the practical reasons for its undecidability.

As shown by Thiemann and Vasconcelos [51], support for polymorphism and polymorphic recursion is paramount in practical applications of context-free session types. Exploring the interaction between polymorphism and subtyping in the context-free setting, possibly in the form of *bounded quantification*, is therefore another avenue for future work.

References

1 Fides Aarts and Frits W. Vaandrager. Learning I/O automata. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010. doi:10.1007/978-3-642-15375-4_6.

2 Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic lambda calculus with context-free session types. *Inf. Comput.*, 289(Part):104948, 2022. doi:10.1016/j.ic.2022.104948.

3 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. FreeST: Context-free session types in a functional language. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 12–23, 2019. doi:10.4204/EPTCS.291.2.

4 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Deciding the bisimilarity of context-free session types. In *TACAS@ 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2020. doi:10.1007/978-3-030-45237-7_3.

5 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Kind inference for the FreeST programming language. *CoRR*, abs/2304.06396, 2023. doi:10.48550/arXiv.2304.06396.

6 Bernardo Almeida, Andreia Mordido, and Vasco Thudichum Vasconcelos. FreeST, a concurrent programming language with context-free session types, 2019. URL: https://freest-lang.github.io/.

7 Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998. doi:10.1007/BFb0055622.

8 Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. doi:10.1145/155183.155231.

9 Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *J. ACM*, 40(3):653–682, 1993. doi:10.1145/174130.174141.

10 Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998. doi:10.3233/FI-1998-33401.

11 Olaf Burkart, Didier Caucal, and Bernhard Steffen. An elementary bisimulation decision procedure for arbitrary context-free processes. In Jirí Wiedermann and Petr Hájek, editors, *Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, August 28 - September 1, 1995, Proceedings*, volume 969 of *Lecture Notes in Computer Science*, pages 423–433. Springer, 1995. doi:10.1007/3-540-60246-1_148.

12 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007. doi:10.1007/978-3-540-71316-6_2.

13 Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, pages 198–199. ACM, 2005. doi:10.1145/1069774.1069793.

14 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 5–17, 2014. doi:10.1145/2535838.2535840.

15 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.

16 Diana Costa, Andreia Mordido, Diogo Poças, and Vasco T. Vasconcelos. Higher-order context-free session types in system F. In Marco Carbone and Rumyana Neykova, editors, *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022*, volume 356 of *EPTCS*, pages 24–35, 2022. doi:10.4204/EPTCS.356.3.

17 Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In *10th International Conference on Mathematics of Program Construction (MPC 2010)*, pages 100–118, Québec City, Canada, June 2010. Springer LNCS 6120. doi:10.1007/978-3-642-13321-3_8.

18 Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Nested session types. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 178–206. Springer, 2021. doi:10.1007/978-3-030-72019-3_7.

19 Stephen Dolan. *Algebraic Subtyping: Distinguished Dissertation 2017*. BCS, Swindon, GBR, 2017. URL: https://www.cs.tufts.edu/~nr/cs257/archive/stephen-dolan/thesis.pdf.

20 Ignacio Fábregas, David de Frutos-Escrig, and Miguel Palomino. Non-strongly stable orders also define interesting simulation relations. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, volume 5728 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2009. doi:10.1007/978-3-642-03741-2_16.

21 Emily P. Friedman. The inclusion problem for simple languages. *Theor. Comput. Sci.*, 1(4):297–316, 1976. doi:10.1016/0304-3975(76)90074-8.

22 Simon Gay. Subtyping between standard and linear function types. Technical report, University of Glasgow, 2006.

23 Simon J. Gay. Bounded polymorphism in session types. *Math. Struct. Comput. Sci.*, 18(5):895–930, 2008. doi:10.1017/S0960129508006944.

24 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.

25 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.

26 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

27 Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965. doi:10.1145/321250.321254.

28 Jan Friso Groote and Hans Hüttel. Undecidable equivalences for basic process algebra. *Inf. Comput.*, 115(2):354–371, 1994. doi:10.1006/inco.1994.1101.

29 Patrick Henry and Géraud Sénizergues. Lalblc a program testing the equivalence of dpda's. In Stavros Konstantinidis, editor, *Implementation and Application of Automata*, pages 169–180, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

30 Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theor. Comput. Sci.*, 158(1&2):143–159, 1996. doi:10.1016/0304-3975(95)00064-X.

31 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

32 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

33 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

34 Ross Horne and Luca Padovani. A logical account of subtyping for session types. *CoRR*, abs/2304.06398, 2023. doi:10.48550/arXiv.2304.06398.

35 Petr Jancar and Faron Moller. Techniques for decidability and undecidability of bisimilarity. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 1999. doi:10.1007/3-540-48320-9_5.

36 A. J. Korenjak and John E. Hopcroft. Simple deterministic languages. In *7th Annual Symposium on Switching and Automata Theory, Berkeley, California, USA, October 23-25, 1966*, pages 36–46. IEEE Computer Society, 1966. doi:10.1109/SWAT.1966.22.

37 Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Polarized subtyping. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*, pages 431–461. Springer International Publishing Cham, 2022.

38 Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 203–210. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5119.

39 Barbara Liskov. Keynote address - data abstraction and hierarchy. In Leigh R. Power and Zvi Weiss, editors, *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1987 Addendum, Orlando, Florida, USA, October 4-8, 1987*, pages 17–34. ACM, 1987. doi:10.1145/62138.62141.

40 Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In Andrew Kennedy and Nick Benton, editors, *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, pages 77–88. ACM, 2010. doi:10.1145/1708016.1708027.

41 Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, pages 481–489. William Kaufmann, 1971. URL: http://ijcai.org/Proceedings/71/Papers/044.pdf.

42 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

43 Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, 2019. doi:10.1145/3229062.

44 David Michael Ritchie Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981. doi:10.1007/BFb0017309.

45 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

46 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci.*, 6(5):409–453, 1996. doi:10.1017/s096012950007002x.

47 Diogo Poças, Diana Costa, Andreia Mordido, and Vasco T. Vasconcelos. System F^{μ}_{ω} with context-free session types. In Thomas Wies, editor, *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, volume 13990 of *Lecture Notes in Computer Science*, pages 392–420. Springer, 2023. doi:10.1007/978-3-031-30044-8_15.

48 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. doi:10.1017/CBO9780511777110.

49 Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping context-free session types. *CoRR*, abs/2307.05661, 2023. doi:10.48550/arXiv.2307.05661.

50 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994. doi:10.1007/3-540-58184-7_118.

51 Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 462–475. ACM, 2016. doi:10.1145/2951913.2951926.

52 Rob J. van Glabbeek. The linear time - branching time spectrum II. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993. doi:10.1007/3-540-57208-2_6.

Asymptotic Complexity Estimates for Probabilistic Programs and Their VASS Abstractions

Michal Ajdarów
Masaryk University, Brno, Czech Republic

Antonín Kučera
Masaryk University, Brno, Czech Republic

Abstract

The standard approach to analyzing the asymptotic complexity of probabilistic programs is based on studying the asymptotic growth of certain expected values (such as the expected termination time) for increasing input size. We argue that this approach is not sufficiently robust, especially in situations when the expectations are infinite. We propose new estimates for the asymptotic analysis of probabilistic programs with non-deterministic choice that overcome this deficiency. Furthermore, we show how to efficiently compute/analyze these estimates for selected classes of programs represented as Markov decision processes over vector addition systems with states.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases Probabilistic programs, asymptotic complexity, vector addition systems

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.12

Related Version *Full Version*: https://arxiv.org/abs/2307.04707 [2]

Funding The work is supported by the Czech Science Foundation, Grant No. 21-24711S.

1 Introduction

Vector Addition Systems with States (VASS) [11] are a model for discrete systems with multiple unbounded resources expressively equivalent to Petri nets [20]. Intuitively, a VASS with $d \geq 1$ counters is a finite directed graph where the transitions are labeled by d-dimensional vectors of integers representing *counter updates*. A computation starts in some state for some initial vector of non-negative counter values and proceeds by selecting transitions non-deterministically and performing the associated counter updates. Since the counters cannot assume negative values, transitions that would decrease some counter below zero are disabled.

In program analysis, VASS are used as abstractions for programs operating over unbounded integer variables. Input parameters are represented by initial counter values, and more complicated arithmetical functions, such as multiplication, are modeled by VASS gadgets computing these functions in a weak sense (see, e.g., [17]). Branching constructs, such as **if-then-else**, are usually replaced with non-deterministic choice. VASS are particularly useful for evaluating the *asymptotic complexity* of infinite-state programs, i.e., the dependency of the running time (and other complexity measures) on the size of the program input [21, 22]. Traditional VASS decision problems such as reachability, liveness, or boundedness are computationally hard [9, 18, 19], and other verification problems such as equivalence-checking [12] or model-checking [10] are even undecidable. In contrast to this, decision problems related to the asymptotic growth of VASS complexity measures are solvable with low complexity and sometimes even in *polynomial time* [4, 23, 15, 16, 1]; see [14] for a recent overview.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 12; pp. 12:1–12:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The existing results about VASS asymptotic analysis are applicable to programs with non-determinism (in *demonic* or *angelic* form, see [5]), but cannot be used to analyze the complexity of *probabilistic programs.* This motivates the study of Markov decision process over VASS (VASS MDPs) with both non-deterministic and probabilistic states, where transitions in probabilistic states are selected according to fixed probability distributions. Here, the problems of asymptotic complexity analysis become even more challenging because VASS MDPs subsume infinite-state stochastic models that are notoriously hard to analyze. So far, the only existing result about asymptotic VASS MDP analysis is [3] where the linearity of expected termination time is shown decidable in polynomial time for VASS MDPs with DAG-like MEC decomposition.

Our Contribution: We study the problems of asymptotic complexity analysis for probabilistic programs and their VASS abstractions.

For non-deterministic programs, termination complexity is a function $\mathcal{L}_{\max}$ assigning to every $n \in \mathbb{N}$ the length of the longest computation initiated in a configuration with each counter set to n. A natural way of generalizing this concept to probabilistic programs is to define a function $\mathcal{L}_{\exp}$ such that $\mathcal{L}_{\exp}(n)$ is the maximal *expected length* of a computation initiated in a configuration of size n, where the maximum is taken over all strategies resolving non-determinism. The same approach is applicable to other complexity measures. We show that this natural idea is generally *inappropriate*, especially in situations when $\mathcal{L}_{\exp}(n)$ is *infinite* for a sufficiently large n. By "inappropriate" we mean that this form of asymptotic analysis can be misleading. For example, if $\mathcal{L}_{\exp}(n) = \infty$ for all $n \geq 1$, one may conclude that the computation takes a very long time independently of n. However, this is not necessarily the case, as demonstrated in a simple example of Fig. 1 (we refer to Section 3 for a detailed discussion). Therefore, we propose new notions of *lower/upper/tight complexity estimates* and demonstrate their advantages over the expected values. These notions can be adapted to other models of probabilistic programs, and constitute the main conceptual contribution of our work.

Then, we concentrate on algorithmic properties of the complexity estimates in the setting of VASS MDPs. Our first result concerns *counter complexity.* We show that for every VASS MDP with DAG-like MEC decomposition and every counter c, there are only two possibilities:

- The function n is a *tight estimate* of the asymptotic growth of the maximal c-counter value assumed along a computation initiated in a configuration of size n.
- The function n^2 is a *lower estimate* of the asymptotic growth of the maximal c-counter value assumed along a computation initiated in a configuration of size n.

Furthermore, it is decidable in *polynomial time* which of these alternatives holds.

Since the termination and transition complexities can be easily encoded as the counter complexity for a fresh "step counter", the above result immediately extends also to these complexities. To some extent, this result can be seen as a generalization of the result about termination complexity presented in [3]. See Section 4 for more details.

Our next result is a full classification of asymptotic complexity for one-dimensional VASS MDPs. We show that for every one-dimensional VASS MDP

- the counter complexity is either unbounded or n is a tight estimate;
- termination complexity is either unbounded or one of the functions n, n^2 is a tight estimate.
- transition complexity is either unbounded, or bounded by a constant, or one of the functions n, n^2 is a tight estimate.

Furthermore, it is decidable in *polynomial time* which of the above cases hold.

Since the complexity of the considered problems remains low, the results are encouraging. On the other hand, they require non-trivial insights, indicating that establishing a full and effective classification of the asymptotic complexity of multi-dimensional VASS MDPs is a challenging problem.

Missing proofs can be found in a full version of this paper [2].

2 Preliminaries

We use $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ to denote the sets of non-negative integers, integers, rational numbers, and real numbers. Given a function $f\colon \mathbb{N} \to \mathbb{N}$, we use $O(f)$ and $\Omega(f)$ to denote the sets of all $g\colon \mathbb{N} \to \mathbb{N}$ such that $g(n) \leq a \cdot f(n)$ and $g(n) \geq b \cdot f(n)$ for all sufficiently large $n \in \mathbb{N}$, where a, b are some positive constants. If $h \in O(f)$ and $h \in \Omega(f)$, we write $h \in \Theta(f)$.

Let A be a finite index set. The vectors of $\mathbb{R}^A$ are denoted by bold letters such as $\mathbf{u}, \mathbf{v}, \mathbf{z}, \ldots$. The component of $\mathbf{v}$ of index $i \in A$ is denoted by $\mathbf{v}(i)$. If the index set is of the form $A = \{1, 2, \ldots, d\}$ for some positive integer d, we write $\mathbb{R}^d$ instead of $\mathbb{R}^A$. For every $n \in \mathbb{N}$, we use $\mathbf{n}$ to denote the constant vector where all components are equal to n. The other standard operations and relations on $\mathbb{R}$ such as $+$, $\leq$, or $<$ are extended to $\mathbb{R}^d$ in the component-wise way. In particular, $\mathbf{v} < \mathbf{u}$ if $\mathbf{v}(i) < \mathbf{u}(i)$ for every index i.

A *probability distribution* over a finite set A is a vector $\nu \in [0,1]^A$ such that $\sum_{a \in A} \nu(a) = 1$. We say that ν is *rational* if every $\nu(a)$ is rational, and *Dirac* if $\nu(a) = 1$ for some $a \in A$.

2.1 VASS Markov Decision Processes

▶ **Definition 1.** *Let $d \geq 1$. A d-dimensional* VASS MDP *is a tuple $\mathcal{A} = (Q, (Q_n, Q_p), T, P)$, where*

- *$Q \neq \emptyset$ is a finite set of* states *split into two disjoint subsets Q_n and Q_p of* nondeterministic *and* probabilistic *states,*
- *$T \subseteq Q \times \mathbb{Z}^d \times Q$ is a finite set of* transitions *such that, for every $p \in Q$, the set $Out(p) \subseteq T$ of all transitions of the form $(p, \mathbf{u}, q)$ is non-empty.*
- *P is a function assigning to each $t \in Out(p)$ where $p \in Q_p$ a positive rational probability so that $\sum_{t \in T(p)} P(t) = 1$.*

The encoding size of $\mathcal{A}$ is denoted by $\|\mathcal{A}\|$, where the integers representing counter updates are written in binary and probability values are written as fractions of binary numbers. For every $p \in Q$, we use $In(p) \subseteq T$ to denote the set of all transitions of the form $(q, \mathbf{u}, p)$. The update vector $\mathbf{u}$ of a transition $t = (p, \mathbf{u}, q)$ is also denoted by $\mathbf{u}_t$.

A *finite path* in $\mathcal{A}$ of length $n \geq 0$ is a finite sequence of the form $p_0, \mathbf{u}_1, p_1, \mathbf{u}_2, \ldots, \mathbf{u}_n, p_n$ where $(p_i, \mathbf{u}_{i+1}, p_{i+1}) \in T$ for all $i < n$. We use $len(\alpha)$ to denote the length of α. If there is a finite path from p to q, we say that q is *reachable* from p. An *infinite path* in $\mathcal{A}$ is an infinite sequence $\pi = p_0, \mathbf{u}_1, p_1, \mathbf{u}_2, \ldots$ such that every finite prefix of π ending in a state is a finite path in $\mathcal{A}$.

A *strategy* is a function σ assigning to every finite path $p_0, \mathbf{u}_1, \ldots, p_n$ such that $p_n \in Q_n$ a probability distribution over $Out(p_n)$. A strategy is *Markovian (M)* if it depends only on the last state p_n, and *deterministic (D)* if it always returns a Dirac distribution. The set of all strategies is denoted by $\Sigma_{\mathcal{A}}$, or just Σ when $\mathcal{A}$ is understood. Every initial state $p \in Q$ and every strategy σ determine the probability space over infinite paths initiated in p in the standard way. We use $\mathbb{P}^\sigma_p$ to denote the associated probability measure.

A *configuration* of $\mathcal{A}$ is a pair $p\mathbf{v}$, where $p \in Q$ and $\mathbf{v} \in \mathbb{Z}^d$. If some component of $\mathbf{v}$ is negative, then $p\mathbf{v}$ is *terminal*. The set of all configurations of $\mathcal{A}$ is denoted by $C(\mathcal{A})$.

```
input $N$
repeat
    random choice:
        $0.5$ :   $N := N + 1$;
        $0.5$ :   $N := N - 1$;
until $N = 0$
```

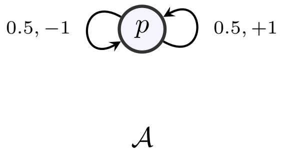


Figure 1 A probabilistic program with infinite expected running time for every $N \geq 1$, and its 1-dimensional VASS MDP model $\mathcal{A}$.

Every infinite path $p_0, \mathbf{u}_1, p_1, \mathbf{u}_2, \ldots$ and every initial vector $\mathbf{v} \in \mathbb{Z}^d$ determine the corresponding *computation* of $\mathcal{V}$, i.e., the sequence of configurations $p_0\mathbf{v}_0, p_1\mathbf{v}_1, p_2\mathbf{v}_2, \ldots$ such that $\mathbf{v}_0 = \mathbf{v}$ and $\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{u}_{i+1}$. Let $\mathit{Term}(\pi)$ be the least j such that $p_j\mathbf{v}_j$ is terminal. If there is no such j, we put $\mathit{Term}(\pi) = \infty$.

Note that every computation uniquely determines its underlying infinite path. We define the probability space over all computations initiated in a given $p\mathbf{v}$, where the underlying probability measure $\mathbb{P}^{\sigma}_{p\mathbf{v}}$ is obtained from $\mathbb{P}^{\sigma}_{p}$ in an obvious way. For a measurable function X over computations, we use $\mathbb{E}^{\sigma}_{p\mathbf{v}}[X]$ to denote the expected value of X.

3 Asymptotic Complexity Measures for VASS MDPs

In this section, we introduce asymptotic complexity estimates applicable to probabilistic programs with non-determinism and their abstract models (such as VASS MDPs). We also explain their relationship to the standard measures based on the expected values of relevant random variables.

Let us start with a simple motivating example. Consider the simple probabilistic program of Fig. 1. The program inputs a positive integer N and then repeatedly increments/decrements N with probability 0.5 until $N = 0$. One can easily show that for every $N \geq 1$, the program terminates with probability one, and the expected termination time is *infinite*. Based on this, one may conclude that the execution takes a very long time, independently of the initial value of N. However, this conclusion is *not* consistent with practical experience gained from trial runs[1]. The program tends to terminate "relatively quickly" for small N, and the termination time *does* depend on N. Hence, the function assigning ∞ to every $N \geq 1$ is *not* a faithful characterization of the asymptotic growth of termination time. We propose an alternative characterization based on the following observations[2]:

- For every $\varepsilon > 0$, the probability of all runs terminating after more than $n^{2+\varepsilon}$ steps (where n is the initial value of N) approaches *zero* as $n \to \infty$.
- For every $\varepsilon > 0$, the probability of all runs terminating after more than $n^{2-\varepsilon}$ steps (where n is the initial value of N) approaches *one* as $n \to \infty$.

Since the execution time is "squeezed" between $n^{2-\varepsilon}$ and $n^{2+\varepsilon}$ for an arbitrarily small $\varepsilon > 0$ as $n \to \infty$, it can be characterized as "asymptotically quadratic". This analysis is in accordance with experimental outcomes.

[1] For $N = 1$, about 95% of trial runs terminate after at most 1000 iterations of the **repeat-until** loop. For $N = 10$, only about 75% of all runs terminate after at most 1000 iterations, but about 90% of them terminate after at most 10000 iterations.

[2] Formal proofs of these observations are simple; in Section 5, we give a full classification of the asymptotic behaviour of one-dimensional VASS MDPs subsuming the trivial example of Fig. 1.

3.1 Complexity of VASS Runs

We recall the complexity measures for VASS runs used in previous works [4, 23, 15, 16, 1]. These functions can be seen as variants of the standard time/space complexities for Turing machines.

Let $\mathcal{A} = (Q, (Q_n, Q_p), T, P)$ be a d-dimensional VASS MDP, $c \in \{1, \ldots, d\}$, and $t \in T$. For every computation $\pi = p_0\mathbf{v}_0, p_1\mathbf{v}_1, p_2\mathbf{v}_2, \ldots$, we put

$$\begin{aligned}
\mathcal{L}(\pi) &= \mathit{Term}(\pi)\\
\mathcal{C}[c](\pi) &= \sup\{\mathbf{v}_i(c) \mid 0 \leq i < \mathit{Term}(\pi)\}\\
\mathcal{T}[t](\pi) &= \text{the total number of all } 0 \leq i < \mathit{Term}(\pi) \text{ such that } (p_i, \mathbf{v}_{i+1}-\mathbf{v}_i, p_{i+1}) = t
\end{aligned}$$

We refer to the functions $\mathcal{L}$, $\mathcal{C}[c]$, and $\mathcal{T}[t]$ as *termination*, *c-counter*, and *t-transition complexity*, respectively.

Let $\mathcal{F}$ be one of the complexity functions defined above. In VASS abstractions of computer programs, the input is represented by initial counter values, and the input size corresponds to the maximal initial counter value. The existing works on *non-probabilistic* VASS concentrate on analyzing the asymptotic growth of the functions $\mathcal{F}_{\max} : \mathbb{N} \to \mathbb{N}_\infty$ where

$$\mathcal{F}_{\max}(n) \quad = \quad \max\{\mathcal{F}(\pi) \mid \pi \text{ is a computation initiated in } p\mathbf{n} \text{ where } p \in Q\}$$

For VASS MDP, we can generalize $\mathcal{F}_{\max}$ into $\mathcal{F}_{\exp}$ as follows:

$$\mathcal{F}_{\exp}(n) \quad = \quad \max\{\mathbb{E}^{\sigma}_{p\mathbf{n}}[\mathcal{F}] \mid \sigma \in \Sigma_{\mathcal{A}}, p \in Q\}$$

Note that for non-probabilistic VASS, the values of $\mathcal{F}_{\max}(n)$ and $\mathcal{F}_{\exp}(n)$ are the same. However, the function $\mathcal{F}_{\exp}$ suffers from the deficiency illustrated in the motivating example at the beginning of Section 3. To see this, consider the one-dimensional VASS MDP $\mathcal{A}$ modeling the simple probabilistic program (see Fig. 1). For every $n \geq 1$ and the only (trivial) strategy σ, we have that $\mathbb{P}^{\sigma}_{p\mathbf{n}}[\mathit{Term} < \infty] = 1$ and $\mathcal{L}_{\exp}(n) = \infty$. However, the practical experience with trial runs of $\mathcal{A}$ is the same as with the original probabilistic program (see above).

3.2 Asymptotic Complexity Estimates

In this section, we introduce asymptotic complexity estimates allowing for a precise analysis of the asymptotic growth of the termination, c-counter, and t-transition complexity, especially when their expected values are infinite for a sufficiently large input. For the sake of readability, we first present a simplified variant applicable to *strongly connected* VASS MDPs.

Let $\mathcal{F}$ be one of the complexity functions for VASS computations defined in Section 3.1, and let $f : \mathbb{N} \to \mathbb{N}$. We say that f is a *tight estimate of* $\mathcal{F}$ if, for arbitrarily small $\varepsilon > 0$, the value of $\mathcal{F}(n)$ is "squeezed" between $f^{1-\varepsilon}(n)$ and $f^{1+\varepsilon}(n)$ as $n \to \infty$. More precisely, for every $\varepsilon > 0$,

- there exist $p \in Q$ and strategies $\sigma_1, \sigma_2, \ldots$ such that $\liminf_{n\to\infty} \mathbb{P}^{\sigma_n}_{p\mathbf{n}}[\mathcal{F} \geq (f(n))^{1-\varepsilon}] \; = \; 1$;
- for all $p \in Q$ and strategies $\sigma_1, \sigma_2, \ldots$ we have that $\limsup_{n\to\infty} \mathbb{P}^{\sigma_n}_{p\mathbf{n}}[\mathcal{F} \geq (f(n))^{1+\varepsilon}] \; = \; 0$.

The above definition is adequate for strongly connected VASS MDPs because tight estimates tend to exist in this subclass. Despite some effort, we have not managed to construct an example of a strongly connected VASS MDP where an $\mathcal{F}$ with some upper polynomial estimate does *not* have a tight estimate (see Conjecture 3). However, if the underlying graph of $\mathcal{A}$ is *not* strongly connected, then the asymptotic growth of $\mathcal{F}$ can differ for computations visiting a different sequence of maximal end components (MECs) of $\mathcal{A}$, and the asymptotic growth of $\mathcal{F}$ can be "squeezed" between $f^{1-\varepsilon}(n)$ and $f^{1+\varepsilon}(n)$ only for the subset of computations visiting the same sequence of MECs. This explains why we need a more general definition of complexity estimates presented below.

An *end component (EC)* of $\mathcal{A}$ is a pair (C, L) where $C \subseteq Q$ and $L \subseteq T$ such that the following conditions are satisfied:

- $C \neq \emptyset$;
- if $p \in C \cap Q_n$, then at least one outgoing transition of p belongs to L;
- if $p \in C \cap Q_p$, then all outgoing transitions of p belong to L;
- if $(p, \mathbf{u}, q) \in L$, then $p, q \in C$;
- for all $p, q \in C$ we have that q is reachable from p and vice versa.

Note that if (C, L) and (C', L') are ECs such that $C \cap C' \neq \emptyset$, then $(C \cup C', L \cup L')$ is also an EC. Hence, every $p \in Q$ either belongs to a unique *maximal end component* (MEC), or does not belong to any EC. Also observe that each MEC can be seen as a strongly connected VASS MDP. We say that $\mathcal{A}$ has *DAG-like MEC decomposition* if for every pair M, M' of different MECs such that the states of M' are reachable from the states of M we have that the states of M are not reachable from the states of M'.

For every infinite path π of $\mathcal{A}$, let $\mathit{mecs}(\pi)$ be the unique sequence of MECs visited by π. Observe that $\mathit{mecs}(\pi)$ disregards the states that do not belong to any EC; intuitively, this is because the transitions executed in such states do not influence the asymptotic growth of $\mathcal{F}$. Observe that the length of $\mathit{mecs}(\pi)$, denoted by $\mathit{len}(\mathit{mecs}(\pi))$, can be finite or infinite. The first possibility corresponds to the situation when an infinite suffix of π stays within the same MEC. Furthermore, for all $\sigma \in \Sigma$ and $p \in Q$, we have that $\mathbb{P}_p^\sigma[\mathit{len}(\mathit{mecs}) = \infty] = 0$, and the probability $\mathbb{P}_p^\sigma[\mathit{len}(\mathit{mecs}) \geq k]$ decays exponentially in k (these folklore results are easy to prove). All of these notions are lifted to computations in an obvious way.

Observe that if a strategy σ aims at maximizing the growth of $\mathcal{F}$, we can safely assume that σ eventually stays in a *bottom* MEC that cannot be exited (intuitively, σ can always move from a non-bottom MEC to a bottom MEC by executing a few extra transitions that do not influence the asymptotic growth of $\mathcal{F}$, and the bottom MEC may allow increasing $\mathcal{F}$ even further). On the other hand, the maximal asymptotic growth of $\mathcal{F}$ may be achievable along some "minimal" sequence of MECs, and this information is certainly relevant for understanding the behaviour of a given probabilistic program. This leads to the following definition:

▶ **Definition 2.** *A* type *is a finite sequence β of MECs such that $\mathit{mecs}(\pi) = \beta$ for some infinite path π.*

We say that f is a lower estimate *of $\mathcal{F}$* for a type *β if for every $\varepsilon > 0$ there exist $p \in Q$ and a sequence of strategies $\sigma_1, \sigma_2, \ldots$ such that $\mathbb{P}_{p\mathbf{n}}^{\sigma_n}[\mathit{mecs} = \beta] > 0$ for all $n \geq 1$ and*

$$\liminf_{n\to\infty} \ \mathbb{P}_{p\mathbf{n}}^{\sigma_n}[\mathcal{F} \geq (f(n))^{1-\varepsilon} \mid \mathit{mecs}{=}\beta] \ = \ 1\,.$$

Similarly, we say that f is an upper estimate *of $\mathcal{F}$* for a type *β if for every $\varepsilon > 0$, every $p \in Q$, and every sequence of strategies $\sigma_1, \sigma_2, \ldots$ such that $\mathbb{P}_{p\mathbf{n}}^{\sigma_n}[\mathit{mecs} = \beta] > 0$ for all $n \geq 1$ we have that*

$$\limsup_{n\to\infty} \ \mathbb{P}_{p\mathbf{n}}^{\sigma_n}[\mathcal{F} \geq (f(n))^{1+\varepsilon} \mid \mathit{mecs}{=}\beta] \ = \ 0$$

If there is no upper estimate of $\mathcal{F}$ for a type β, we say that $\mathcal{F}$ is unbounded for *β. Finally, we say that f is a* tight estimate *of $\mathcal{F}$* for *β if it is both a lower estimate and an upper estimate of $\mathcal{F}$ for β.*

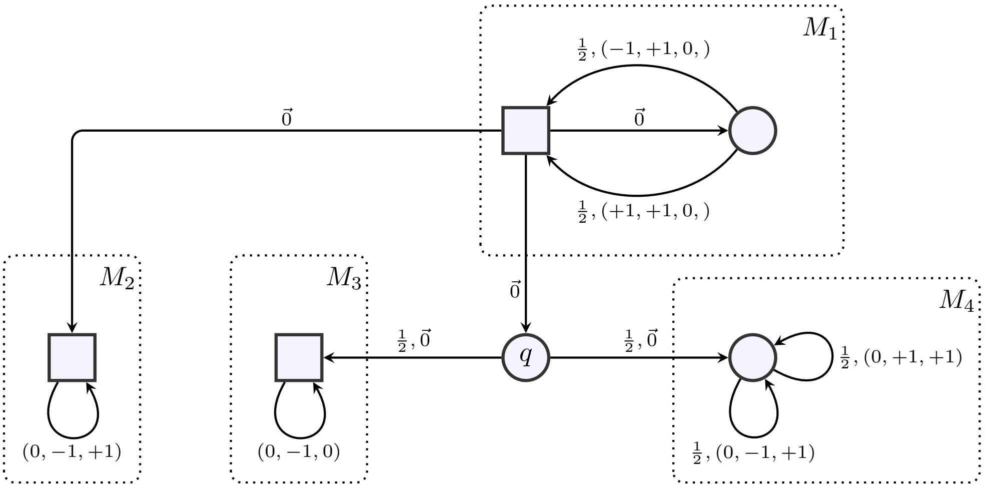


Figure 2 A VASS MDP $\mathcal{A}$ with four MECs and seven types.

Let us note that in the subclass of *non-probabilistic* VASS, MECs become strongly connected components (SCCs), and types correspond to paths in the directed acyclic graph of SCCs. Each such path determines the corresponding asymptotic increase of $\mathcal{F}$, as demonstrated in [1]. We conjecture that types play a similar role for VASS MDPs. More precisely, we conjecture the following:

▶ **Conjecture 3.** *If some polynomial is an upper estimate of $\mathcal{F}$ for β, then there exists a tight estimate f of $\mathcal{F}$ for β.*

Even if Conjecture 3 is proven wrong, there are interesting subclasses of VASS MDPs where it holds, as demonstrated in subsequent sections.

For every pair of MECs M, M', let $P(M, M')$ be the maximal probability (achievable by some strategy) of reaching a state of M' from a state of M in $\mathcal{A}$ without passing through a state of some other MEC M''. Note that $P(M, M')$ is efficiently computable by standard methods for finite-state MDPs. The *weight* of a given type $\beta = M_1, \ldots, M_k$ is defined as $\mathit{weight}(\beta) = \prod_{i=1}^{k-1} P(M_i, M_{i+1})$. Intuitively, $\mathit{weight}(\beta)$ corresponds to the maximal probability of "enforcing" the asymptotic growth of $\mathcal{F}$ according to the tight estimate f of $\mathcal{F}$ for β achievable by some strategy.

Generally, higher asymptotic growth of $\mathcal{F}$ may be achievable for types with smaller weights. Consider the following example to understand better the types, their weights, and the associated tight estimates.

▶ **Example 4.** Let $\mathcal{A}$ be the VASS MDP of Fig. 2. There are four MECs M_1, M_2, M_3, M_4 where M_2, M_3, M_4 are bottom MECs. Hence, there are four types of length one and three types of length two. Let us examine the types of length two initiated in M_1 for $\mathcal{F} \equiv \mathcal{C}[c]$ where c is the third counter.

Note that in M_1, the first counter is repeatedly incremented/decremented with the same probability $\frac{1}{2}$. The second counter "counts" these transitions and thus it is "pumped" to a *quadratic* value (cf. the VASS MDP of Fig. 1). Then, a strategy may decide to move to M_2, where the value of the second counter is transferred to the third counter. Hence, n^2 is the tight estimate of $\mathcal{C}[c]$ for the type M_1, M_2, and $\mathit{weight}(M_1, M_2) = 1$. Alternatively, a strategy may decide to move to the probabilistic state q. Then, either M_3 or M_4 is entered with the same probability $\frac{1}{2}$, which implies $\mathit{weight}(M_1, M_3) = \mathit{weight}(M_1, M_4) = \frac{1}{2}$. In M_3, the third counter is unchanged, and hence n is the tight estimate of $\mathcal{C}[c]$ for the type M_1, M_3. However, in M_4, the second counter previously pumped to a quadratic value is repeatedly incremented/decremented with the same probability $\frac{1}{2}$, and the third counter "counts" these transitions. This means that n^4 is a tight estimate of $\mathcal{C}[c]$ for the type M_1, M_4.

This analysis provides detailed information about the asymptotic growth of $\mathcal{C}[c]$ in $\mathcal{A}$. Every type shows "how" the growth specified by the corresponding tight estimate is achievable, and its weight corresponds to the "maximal achievable probability of this growth". This information is completely lost when analyzing the maximal expected value of $\mathcal{C}[c]$ for computations initiated in configurations $p\mathbf{n}$ where p is a state of M_1, because these expectations are *infinite* for all $n \geq 1$.

Finally, let us clarify the relationship between the lower/upper estimates of $\mathcal{F}$ and the asymptotic growth of $\mathcal{F}_{\mathrm{exp}}$. The following observation is easy to prove.

▶ **Observation 5.** *If $\mathcal{F}_{\mathrm{exp}} \in O(f)$ where $f : \mathbb{N} \to \mathbb{N}$ is an unbounded function, then f is an upper estimate of $\mathcal{F}$ for every type. Furthermore, if $f : \mathbb{N} \to \mathbb{N}$ is a lower estimate of $\mathcal{F}$ for some type, then $\mathcal{F}_{\mathrm{exp}} \in \Omega(f^{1-\epsilon})$ for each $\epsilon > 0$. However, if $\mathcal{F}_{\mathrm{exp}} \in \Omega(f)$ where $f : \mathbb{N} \to \mathbb{N}$, then f is* not *necessarily a lower estimate of $\mathcal{F}$ for some type.*

Observation 5 shows that complexity estimates are generally more informative than the asymptotics of $\mathcal{F}_{\mathrm{exp}}$ even if $\mathcal{F}_{\mathrm{exp}} \in \Theta(f)$ for some "reasonable" function f. For example, it may happen that there are only two types β_1 and β_2 where n and n^3 are tight estimates of $\mathcal{L}$ for β_1 and β_2 with weights 0.99 and 0.01, respectively. In this case, $\mathcal{L}_{\mathrm{exp}} \in \Theta(n^3)$, although the termination time is linear for 99% of computations.

4 A Dichotomy between Linear and Quadratic Estimates

In this section, we prove the following result:

▶ **Theorem 6.** *Let $\mathcal{A}$ be a VASS MDP with DAG-like MEC decomposition and $\mathcal{F}$ one of the complexity functions $\mathcal{L}$, $\mathcal{C}[c]$, or $\mathcal{T}[t]$. For every type β, we have that either n is a tight estimate of $\mathcal{F}$ for β, or n^2 is a lower estimate of $\mathcal{F}$ for β. It is decidable in polynomial time which of the two cases holds.*

Theorem 6 can be seen as a generalization of the linear/quadratic dichotomy results previously achieved for non-deterministic VASS [4] and for the termination complexity in VASS MDPs [3].

It suffices to prove Theorem 6 for the *counter complexity*. The corresponding results for the termination and transition complexities then follow as simple consequences. To see this, observe that we can extend a given VASS MDP with a fresh "step counter" sc that is incremented by every transition (in the case of $\mathcal{L}$) or the transition t (in the case of $\mathcal{T}[t]$) and thus "emulate" $\mathcal{L}$ and $\mathcal{T}[t]$ as $\mathcal{C}[sc]$.

We first consider the case when $\mathcal{A}$ is strongly connected and then generalize the obtained results to VASS MDPs with DAG-like MEC decomposition. So, let $\mathcal{A}$ be a strongly connected d-dimensional VASS MDP and c a counter of $\mathcal{A}$. The starting point of our analysis is the dual constraint system designed in [23] for non-probabilistic strongly connected VASS. We generalize this system to strongly connected VASS MDPs in the way shown in Figure 3 (the original system of [23] can be recovered by disregarding the probabilistic states).

Note that solutions of both (I) and (II) are closed under addition. Therefore, both (I) and (II) have solutions maximizing the specified objectives, computable in polynomial time. For clarity, let us first discuss an intuitive interpretation of these solutions, starting with simplified variants obtained for non-probabilistic VASS in [23].

Constraint system (I):

Find $\mathbf{x} \in \mathbb{Z}^T$ such that

$$\sum_{t \in T} \mathbf{x}(t)\mathbf{u}_t \geq \vec{0}$$

$$\mathbf{x} \geq \vec{0}$$

and for each $p \in Q$

$$\sum_{t \in Out(p)} \mathbf{x}(t) = \sum_{t \in In(p)} \mathbf{x}(t)$$

and for all $p \in Q_p$, $t \in Out(p)$

$$\mathbf{x}(t) = P(t) \cdot \sum_{t' \in Out(p)} \mathbf{x}(t')$$

Objective: *Maximize*

- the number of valid inequalities of the form
$$\sum_{t \in T} \mathbf{x}(t)\mathbf{u}_t(c) > 0,$$
- the number of valid inequalities of the form $\mathbf{x}(t) > 0$.

Constraint system (II):

Find $\mathbf{y} \in \mathbb{Z}^d, \mathbf{z} \in \mathbb{Z}^Q$ such that

$$\mathbf{y} \geq \vec{0}$$

$$\mathbf{z} \geq \vec{0}$$

and for each $(p, \mathbf{u}, q) \in T$ where $p \in Q_n$

$$\mathbf{z}(q) - \mathbf{z}(p) + \sum_{i=1}^{d} \mathbf{u}(i)\mathbf{y}(i) \leq 0$$

and for each $p \in Q_p$

$$\sum_{t=(p,\mathbf{u},q) \in Out(p)} P(t)\big(\mathbf{z}(q) - \mathbf{z}(p) + \sum_{i=1}^{d} \mathbf{u}_t(i)\mathbf{y}(i)\big) \leq 0$$

Objective: *Maximize*

- the number of valid inequalities of the form $\mathbf{y}(c) > 0$,
- the number of transitions $t = (p, \mathbf{u}, q)$ such that $p \in Q_n$ and
$$\mathbf{z}(q) - \mathbf{z}(p) + \sum_{i=1}^{d} \mathbf{u}(i)\mathbf{y}(i) < 0,$$
- the number of states $p \in Q_p$ such that
$$\sum_{t=(p,\mathbf{u},q) \in Out(p)} P(t)\big(\mathbf{z}(q) - \mathbf{z}(p) + \sum_{i=1}^{d} \mathbf{u}(i)\mathbf{y}(i)\big) < 0\,.$$

Figure 3 Constraint systems for strongly connected VASS MDPs.

In the non-probabilistic case, a solution of (I) can be interpreted as a *weighted multicycle*, i.e., as a collection of cycles $M_1, \ldots, M_k$ together with weights $a_1, \ldots, a_k$ such that the total effect of the multicycle, defined by $\sum_{i=1}^{k} a_i \cdot \mathit{effect}(M_i)$, is non-negative for every counter. Here, $\mathit{effect}(M_i)$ is the effect of M_i on the counters. The objective of (I) ensures that the multicycle includes as many transitions as possible, and the total effect of the multicycle is positive on as many counters as possible. For VASS MDPs, the $M_1, \ldots, M_k$ should not be interpreted as cycles but as Markovian strategies for some ECs, and $\mathit{effect}(M_i)$ corresponds to the vector of expected counter changes per transition in M_i. The objective of (I) then maximizes the number of transitions used in the strategies $M_1, \ldots, M_k$, and the number of counters where the expected effect of the "multicycle" is positive.

A solution of (II) for non-probabilistic VASS can be interpreted as a ranking function for configurations defined by $\mathit{rank}(p\mathbf{v}) = \mathbf{z}(p) + \sum_{i=1}^{d} \mathbf{y}(i)\mathbf{v}(i)$, such that the value of *rank* cannot increase when moving from a configuration $p\mathbf{v}$ to a configuration $q\mathbf{u}$ using a transition $t = (p, \mathbf{u} - \mathbf{v}, q)$. The objective of (II) ensures that as many transitions as possible decrease the value of *rank*, and *rank* depends on as many counters as possible. For VASS MDPs, this interpretation changes only for the outgoing transitions $t = (p, \mathbf{u}, q)$ of probabilistic states. Instead of considering the change of *rank* caused by such t, we now consider the expected change of *rank* caused by executing a step from p. The objective ensures that *rank* depends on as many counters as possible, the value of *rank* is decreased by as many outgoing transitions of non-deterministic states as possible, and the expected change of *rank* caused by performing an step is negative in as many probabilistic states as possible.

The key tool for our analysis is the following dichotomy:

▶ **Lemma 7.** *Let* $\mathbf{x}$ *be a (maximal) solution to the constraint system (I) and* $\mathbf{y}, \mathbf{z}$ *be a (maximal) solution to the constraint system (II). Then, for each counter* c *we have that either* $\mathbf{y}(c) > 0$ *or* $\sum_{t \in T} \mathbf{x}(t)\mathbf{u}_t(c) > 0$*, and for each transition* $t = (p, \mathbf{u}, q) \in T$ *we have that*

- *if* $p \in Q_n$ *then either* $\mathbf{z}(q) - \mathbf{z}(p) + \sum_{i=1}^{d} \mathbf{u}(i)\mathbf{y}(i) < 0$ *or* $\mathbf{x}(t) > 0$*;*
- *if* $p \in Q_p$ *then either*

$$\sum_{t'=(p,\mathbf{u}',q') \in Out(p)} P(t')\big(\mathbf{z}(q') - \mathbf{z}(p) + \sum_{i=1}^{d} \mathbf{u}'(i)\mathbf{y}(i)\big) < 0$$

or $\mathbf{x}(t) > 0$*.*

For the rest of this section, we fix a maximal solution $\mathbf{x}$ of (I) and a maximal solution $\mathbf{y}, \mathbf{z}$ of (II), such that the smallest non-zero element of $\mathbf{y}, \mathbf{z}$ is at least 1. We define a ranking function $rank : C(\mathcal{A}) \to \mathbb{N}$ as $rank(s\mathbf{v}) = \mathbf{z}(s) + \sum_{i=1}^{d} \mathbf{v}(i)\mathbf{y}(i)$.

▶ **Theorem 8.** *For each counter* c*, if* $\mathbf{y}(c) > 0$ *then* n *is a tight estimate of* $\mathcal{C}[c]$ *(for the only type of* $\mathcal{A}$*). Otherwise, i.e., when* $\mathbf{y}(c) = 0$*, the function* n^2 *is a lower estimate of* $\mathcal{C}[c]$*.*

Note that Theorem 8 implies Theorem 6 for strongly connected VASS MDPs. A proof is obtained by combining the following lemmata.

▶ **Lemma 9.** *For every counter* c *such that* $\mathbf{y}(c) > 0$*, every* $\varepsilon > 0$*, every* $p \in Q$*, and every* $\sigma \in \Sigma$*, there exists* n_0 *such that for all* $n \geq n_0$ *we have that* $\mathbb{P}^{\sigma}_{p\mathbf{n}}(\mathcal{C}[c] \geq n^{1+\varepsilon}) \leq kn^{-\varepsilon}$ *where* k *is a constant depending only on* $\mathcal{A}$*.*

For *Targets* $\subseteq C(\mathcal{A})$ and $m \in \mathbb{N}$, we use $\mathrm{Reach}^{\leq m}(\mathit{Targets})$ to denote the set of all computations $\pi = p_0\mathbf{v}_0, p_1\mathbf{v}_1, \ldots$ such that $p_i\mathbf{v}_i \in \mathit{Targets}$ for some $i \leq m$.

▶ **Lemma 10.** *For each counter* c *such that* $\mathbf{y}(c) = 0$ *we have that* $\mathcal{C}_{\exp}[c] \in \Omega(n^2)$ *and* n^2 *is a lower estimate of* $\mathcal{C}[c]$*. Furthermore, for every* $\varepsilon > 0$ *there exist a sequence of strategies* $\sigma_1, \sigma_2, \ldots$*, a constant* k*, and* $p \in Q$ *such that for every* $0 < \varepsilon' < \varepsilon$*, we have that*

$$\lim_{n\to\infty} \mathbb{P}^{\sigma_n}_{p\mathbf{n}}(\mathrm{Reach}^{\leq kn^{2-\varepsilon'}}(\mathit{Targets}_n)) = 1$$

where $\mathit{Targets}_n = \{q\mathbf{v} \in C(\mathcal{A}) \mid \mathbf{v}(c) \geq n^{2-\varepsilon}$ *for every counter* c *such that* $\mathbf{y}(c) = 0\}$*.*

It remains to prove Theorem 6 for VASS MDPs with DAG-like MEC decomposition. Here, we proceed by analyzing the individual MECs one by one, transferring the output of the previous MEC to the next one. We start in a top MEC with all counters initialized to n. Here we can directly apply Theorem 8 to determine which of the $\mathcal{C}[c]$ have a tight estimate n and a lower estimate n^2, respectively. It follows from Lemma 10 that all counters c such that n^2 is a lover estimate of $\mathcal{C}[c]$ can be simultaneously pumped to $n^{2-\varepsilon}$ with very high probability. However, this computation may decrease the counters c such that n is a tight estimate for $\mathcal{C}[c]$. To ensure that the value of these counters is still $\Omega(n)$ when entering the next MEC, we first divide the initial counter vector $\mathbf{n}$ into two halves, each of size $\lfloor \frac{\mathbf{n}}{2} \rfloor$, and then pump the counters c such that n^2 is a lower estimate for $\mathcal{C}[c]$ to the value $(\lfloor \frac{n}{2} \rfloor)^{2-\varepsilon}$. We show that the length of this computation is at most quadratic. The value of the other counters stays at least $\lfloor \frac{n}{2} \rfloor$. When analyzing the next MEC, we treat the counters previously pumped to quadratic values as "infinite" because they are sufficiently large so that they cannot prevent pumping additional counters to asymptotically quadratic values. Technically, this is implemented by modifying every counter update vector $\mathbf{u}$ so that $\mathbf{u}[c] = 0$ for every "quadratic" counter c. A precise formulation of these observations and the corresponding proofs are given in [2].

We conjecture that the dichotomy of Theorem 6 holds for *all* VASS MDPs, but we do not have a complete proof. If the MEC decomposition is not DAG-like, a careful analysis of computations revisiting the same MECs is required; such repeated visits may but do not have to enable additional asymptotic growth of $\mathcal{C}[c]$.

5 One-Dimensional VASS MDPs

In this section, we give a full and effective classification of tight estimates of $\mathcal{L}$, $\mathcal{C}[c]$, and $\mathcal{T}[t]$ for one-dimensional VASS MDPs. More precisely, we prove the following theorem:

▶ **Theorem 11.** *Let $\mathcal{A}$ be a one-dimensional VASS MDP. We have the following:*

- *Let c be the only counter of $\mathcal{A}$. Then one of the following possibilities holds:*
  - *There exists a type $\beta = M$ such that $\mathcal{C}[c]$ is unbounded for β.*
  - *n is a tight estimate of $\mathcal{C}[c]$ for every type.*
- *Let t be a transition of $\mathcal{A}$. Then one of the following possibilities holds:*
  - *There exists a type $\beta = M$ such that $\mathcal{T}[t]$ is unbounded for β.*
  - *There exists a type β such that weight$(\beta) > 0$ and $\mathcal{T}[t]$ is unbounded for β.*
  - *There exists a type $\beta = M$ such that n^2 is a tight estimate of $\mathcal{T}[t]$ for β.*
  - *The transition t occurs in some MEC M, n is a tight estimate of $\mathcal{T}[t]$ for every type β containing the MEC M, and 0 is a tight estimate of $\mathcal{T}[t]$ for every type β not containing the MEC M.*
  - *The transition t does not occur in any MEC, and for every type β of length k we have that k is an upper estimate of $\mathcal{T}[t]$ for β.*
- *One of the following possibilities holds:*
  - *There exists a type $\beta = M$ such that $\mathcal{L}$ is unbounded for β.*
  - *There exists a type $\beta = M$ such that n^2 is a tight estimate of $\mathcal{L}$ for β.*
  - *n is a tight estimate of $\mathcal{L}$ for every type.*

It is decidable in polynomial time which of the above cases hold.

Note that some cases are mutually exclusive and some may hold simultaneously. Also recall that *weight*$(\beta) = 1$ for every type β of length one, and *weight*(β) decays exponentially in the length of β. Hence, if a transition t does not occur in any MEC, there is a constant $\kappa < 1$ depending only on $\mathcal{A}$ such that $\mathbb{P}^{\sigma}_{p\mathbf{v}}[\mathcal{T}[t] \geq i] \leq \kappa^i$ for every $\sigma \in \Sigma$ and $p\mathbf{v} \in C(\mathcal{A})$.

For the rest of this section, we fix a one-dimensional VASS MDP $\mathcal{A} = (Q, (Q_n, Q_p), T, P)$ and some linear ordering $\sqsubseteq$ on Q. A proof of Theorem 11 is obtained by analyzing bottom strongly connected components (BSCCs) in a Markov chain obtained from $\mathcal{A}$ by "applying" some MD strategy σ (we use Σ_{MD} to denote the class of all MD strategies for $\mathcal{A}$). Recall that σ selects the same outgoing transition in every $p \in Q_n$ whenever p is revisited, and hence we can "apply" σ to $\mathcal{A}$ by removing the other outgoing transitions. The resulting Markov chain is denoted by $\mathcal{A}_\sigma$. Note that every BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ can also be seen as an end component of $\mathcal{A}$. For a MEC M of $\mathcal{A}$, we write $\mathbb{B} \subseteq M$ if all states and transitions of $\mathbb{B}$ are included in M.

For every BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$, let $p_{\mathbb{B}}$ be the least state of $\mathbb{B}$ with respect to $\sqsubseteq$. Let $\mathbb{U}_{\mathbb{B}}$ be a function assigning to every infinite path $\pi = p_0, \mathbf{u}_1, p_1, \mathbf{u}_2, \ldots$ the sum $\sum_{i=1}^{\ell} \mathbf{u}_i$ if $p_0 = p_{\mathbb{B}}$ and $\ell \geq 1$ is the least index such that $p_\ell = p_{\mathbb{B}}$, otherwise $\mathbb{U}_{\mathbb{B}}(\pi) = 0$. Hence, $\mathbb{U}_{\mathbb{B}}(\pi)$ is the change of the (only) counter c along π until $p_{\mathbb{B}}$ is revisited.

▶ **Definition 12.** *Let $\mathbb{B}$ be a BSCC of $\mathcal{A}_\sigma$. We say that $\mathbb{B}$ is*

- increasing *if* $\mathbb{E}^{\sigma}_{p_{\mathbb{B}}}(\mathbb{U}_B) > 0$*,*
- decreasing *if* $\mathbb{E}^{\sigma}_{p_{\mathbb{B}}}(\mathbb{U}_B) < 0$*,*
- bounded-zero *if* $\mathbb{E}^{\sigma}_{p_{\mathbb{B}}}(\mathbb{U}_B) = 0$ *and* $\mathbb{P}^{\sigma}_{p_{\mathbb{B}}}[\mathbb{U}_{\mathbb{B}}{=}0] = 1$*,*
- unbounded-zero *if* $\mathbb{E}^{\sigma}_{p_{\mathbb{B}}}(\mathbb{U}_B) = 0$ *and* $\mathbb{P}^{\sigma}_{p_{\mathbb{B}}}[\mathbb{U}_{\mathbb{B}}{=}0] < 1$*.*

Note that the above definition does not depend on the concrete choice of $\sqsubseteq$. We prove the following results relating the existence of upper/lower estimates of $\mathcal{L}$, $\mathcal{C}[c]$, and $\mathcal{T}[t]$ to the existence of BSCCs with certain properties. More concretely,

- for $\mathcal{C}[c]$, we show that
  - $\mathcal{C}[c]$ is unbounded for some type $\beta = M$ if there exists an increasing BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$ such that $\mathbb{B} \subseteq M$;
  - otherwise, n is a tight estimate of $\mathcal{C}[c]$ for every type.
- for $\mathcal{L}$, we show that
  - $\mathcal{L}$ is unbounded for some type $\beta = M$ if there exists an increasing or bounded-zero BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$ such that $\mathbb{B} \subseteq M$;
  - otherwise, n^2 is an upper estimate of $\mathcal{L}$ for every type β;
  - if there exists an unbounded-zero BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$, then n^2 is a lower estimate of $\mathcal{L}$ for $\beta = M$ where $\mathbb{B} \subseteq M$;
  - if every BSCC $\mathbb{B}$ of every $\mathcal{A}_\sigma$ is decreasing, then $\mathcal{L}_{\exp}(n) \in \Theta(n)$ (this follows from [3]), and hence n is a tight estimate of $\mathcal{L}$ for every type (Observation 5);
- for $\mathcal{T}[t]$, we distinguish two cases:
  - If t is not contained in any MEC of $\mathcal{A}$, then for every type β of length k, the transition t cannot be executed more than k times along a arbitrary computation π where $\mathit{mecs}(\pi) = \beta$.
  - If t is contained in a MEC M of $\mathcal{A}$, then
    - $\mathcal{T}[t]$ is unbounded for $\beta = M$ if there exist an increasing BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$ such that $\mathbb{B} \subseteq M$, or bounded-zero BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$ such that $\mathbb{B}$ contains t;
    - $\mathcal{T}[t]$ is unbounded for every $\beta = M_1, \ldots, M_k$ such that $M = M_i$ for some i and there exists an increasing BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$ such that $\mathbb{B} \subseteq M_j$ for some $j \leq i$;
    - otherwise, n^2 is an upper estimate of $\mathcal{T}[t]$ for every type;
    - if there is an unbounded-zero BSCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$ such that $\mathbb{B}$ contains t, then n^2 is a lower estimate of $\mathcal{T}[t]$ for $\beta = M$;
    - if every BSCC $\mathbb{B}$ of every $\mathcal{A}_\sigma$ is decreasing, then $\mathcal{T}[t]_{\exp}(n) \in \Theta(n)$ (this follows from [3]), and hence n is an upper estimate of $\mathcal{T}[t]$ for every type (Observation 5).

The polynomial time bound of Theorem 11 is then obtained by realizing the following: First, we need to decide the existence of an increasing BSCC of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$. This can be done in polynomial time using the constraint system (I) of Figure 3. If no such increasing BSCC exists, we need to decide the existence of a bounded-zero BSCC, which can be achieved in polynomial time for a subclass of one-dimensional VASS MDPs where no increasing BSCC exists. Then, if no bounded-zero BSCC exists, we need to decide the existence of an unbounded-zero BSCC, which can again be done in polynomial time using the constraint system (I) of Figure 3 (realize that any solution $\mathbf{x}$ of (I) implies the existence of a BSCC that is either increasing, bounded-zero, or unbounded-zero).

Hence, the "algorithmic part" of Theorem 11 is an easy consequence of the above observations, but there is one remarkable subtlety. Note that we need to decide the existence of a bounded-zero BSCC only for a subclass of one-dimensional VASS MDPs where no increasing BSCCs exist. This is actually crucial, because deciding the existence of a bounded-zero BSCC in *general* one-dimensional VASS MDPs is **NP**-complete [2].

The main difficulties requiring novel insights are related to proving the observation about $\mathcal{C}[c]$, stating that if there is no increasing BSCC of $\mathcal{A}_\sigma$ for any $\sigma \in \Sigma_{\text{MD}}$, then n is an upper estimate of $\mathcal{C}[c]$ for every type. A comparably difficult (and in fact closely related) task is to show that if there is no increasing or bounded-zero BSCC, then n^2 is an upper estimate of $\mathcal{L}$ for every type. Note that here we need to analyze the behaviour of $\mathcal{A}$ under *all* strategies (not just MD), and consider the notoriously difficult case when the long-run average change of the counter caused by applying the strategy is zero. Here we need to devise a suitable decomposition technique allowing for interpreting general strategies as "interleavings" of MD strategies and lifting the properties of MD strategies to general strategies. Furthermore, we need to devise techniques for reducing the problems of our interest to analyzing certain types of random walks that have already been studied in stochastic process theory. We discuss this more in the following subsection, and we refer to [2] for a complete exposition of these results.

5.1 MD decomposition

As we already noted, one crucial observation behind Theorem 11 is that if there is no increasing BSCC of $\mathcal{A}_\sigma$ for any $\sigma \in \Sigma_{\text{MD}}$, then n is an upper estimate of $\mathcal{C}[c]$ for every type. In this section, we sketch the main steps towards this result.

First, we show that every path in $\mathcal{A}$ can be decomposed into "interweavings" of paths generated by MD strategies.

Let $\alpha = p_0, \mathbf{v}_1, \ldots, p_k$ be a path. For every $i \leq k$, we use $\alpha_{..i} = p_0, \mathbf{v}_1, \ldots, p_i$ to denote the prefix of α of length i. We say that α is *compatible* with a MD strategy σ if $\sigma(\alpha_{..i}) = (p_i, \mathbf{v}_{i+1}, p_{i+1})$ for all $i < k$ such that $p_i \in Q_n$. Furthermore, for every path $\beta = q_0, \mathbf{u}_1, q_1, \ldots, q_\ell$ such that $p_k = q_0$, we define a path $\alpha \circ \beta = p_0, \mathbf{v}_1, p_1, \ldots, p_k, \mathbf{u}_1, q_1, \ldots, q_\ell$.

▶ **Definition 13.** *Let $\mathcal{A}$ be a VASS MDP, $\pi_1, \ldots, \pi_k \in \Sigma_{\text{MD}}$, and $p_1, \ldots, p_k \in Q$. An* MD-decomposition *of a path $\alpha = s_1, \ldots, s_m$ under $\pi_1, \ldots, \pi_k$ and $p_1, \ldots, p_k$ is a decomposition of α into finitely many paths $\alpha = \gamma_1^1 \circ \cdots \circ \gamma_1^k \circ \gamma_2^1 \circ \cdots \circ \gamma_2^k \circ \cdots \circ \gamma_\ell^1 \circ \cdots \circ \gamma_\ell^k$ satisfying the following conditions:*

- *for all $i < \ell$ and $j \leq k$, the last state of γ_i^j is the same as the first state of γ_{i+1}^j;*
- *for every $j \leq k$, $\gamma_1^j \circ \cdots \circ \gamma_\ell^j$ is a path that begins with p_j and is compatible with π_j.*

Note that $\pi_1, \ldots, \pi_k$ and $p_1, \ldots, p_k$ are not necessarily pairwise different, and the length of γ_i^j can be zero. Also note that the same α may have several MD-decompositions.

Intuitively, an MD decomposition of α shows how to obtain α by repeatedly selecting zero or more transitions by $\pi_1, \ldots, \pi_k$. The next lemma shows that for every VASS MDP $\mathcal{A}$, one can *fix* MD strategies $\pi_1, \ldots, \pi_k$ and states $p_1, \ldots, p_k$ such that *every* path α in $\mathcal{A}$ has an MD-decomposition under $\pi_1, \ldots, \pi_k$ and $p_1, \ldots, p_k$. Furthermore, such a decomposition is constructible *online* as α is read from left to right.

▶ **Lemma 14.** *For every VASS MDP $\mathcal{A}$, there exist $\pi_1, \ldots, \pi_k \in \Sigma_{\text{MD}}$, $p_1, \ldots, p_k \in Q$, and a function $Decomp_{\mathcal{A}}$ such that the following conditions are satisfied for every finite path α:*

- *$Decomp_{\mathcal{A}}(\alpha)$ returns an MD-decomposition of α under $\pi_1, \ldots, \pi_k$ and $p_1, \ldots, p_k$.*
- *$Decomp_{\mathcal{A}}(\alpha) = Decomp_{\mathcal{A}}(\alpha_{..len(\alpha)-1}) \circ \gamma^1 \circ \cdots \circ \gamma^k$, where exactly one of γ^i has positive length (the i is called the* mode *of α).*
- *If the last state of $\alpha_{..len(\alpha)-1}$ is probabilistic, then the mode of α does not depend on the last transition of α.*

According to Lemma 14, *every* strategy σ for $\mathcal{A}$ just performs a certain "interleaving" of the MD strategies $\pi_1, \ldots, \pi_k$ initiated in the states $p_1, \ldots, p_k$. We aim to show that if every BSCC of every $\mathcal{A}_{\pi_j}$ is non-increasing, then n is an upper estimate of $\mathcal{C}[c]$ for every type. Since we do not have any control over the length of the individual γ_i^j occurring in MD-decompositions, we need to introduce another concept of *extended VASS MDPs* where the strategies $\pi_1, \ldots, \pi_k$ can be interleaved in "longer chunks". Intuitively, an extended VASS MDP is obtained from $\mathcal{A}$ by taking k copies of $\mathcal{A}$ sharing the same counter. The j-th copy selects transitions according to π_j. At each round, only one π_j makes a move, where the j is selected by a special type of "pointing" strategy defined especially for extended MDPs. Note that σ can be faithfully simulated in the extended VASS MDP by a pointing strategy that selects the indexes consistently with $Decomp_{\mathcal{A}}$. However, we can also construct another pointing strategy that simulates each π_j longer (i.e., "precomputes" the steps executed by π_j in the future) and thus "close cycles" in the BSCC visited by π_j. This computation can be seen as an interleaving of a finite number of independent random walks with non-positive expectations. Then, we use the optional stopping theorem to get an upper bound on the total expected number of "cycles", which can then be used to obtain the desired upper estimate. We refer to [2] for details.

5.2 A Note about Energy Games

One-dimensional VASS MDPs are closely related to energy games/MDPs [6, 7, 8, 13]. An important open problem for energy games is the complexity of deciding the existence of a *safe* configuration where, for a sufficiently high energy amount, the responsible player can avoid decreasing the energy resource (counter) below zero. This problem is known to be in **NP** ∩ **coNP**, and a pseudopolynomial algorithm for the problem exists; however, it is still open whether the problem is in **P** when the counter updates are encoded in binary. Our analysis shows that this problem is solvable in polynomial time for energy (i.e., one-dimensional VASS) MDPs $\mathcal{A}$ such that there is no increasing SCC of $\mathcal{A}_\sigma$ for any $\sigma \in \Sigma_{\text{MD}}$.

We say that a SCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ is *non-decreasing* if $\mathbb{B}$ does not contain any negative cycles. Note that every bounded-zero SCC is non-decreasing, and a increasing SCC may but does not have to be non-decreasing.

▶ **Lemma 15.** *An energy MDP has a safe configuration iff there exists a non-decreasing SCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\text{MD}}$.*

The "⇐" direction of Lemma 15 is immediate, and the other direction can be proven using our MD decomposition technique, see [2].

Note that if there is no increasing SCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for any $\sigma \in \Sigma_{\text{MD}}$, then the existence of a non-decreasing SCC is equivalent to the existence of a bounded-zero SCC, and hence it can be decided in polynomial time (see the results presented above). However, for general energy MDPs, the best upper complexity bound for the existence of a non-decreasing SCC is **NP** ∩ **coNP**. Interestingly, a small modification of this problem already leads to **NP**-completeness, as demonstrated by the following lemma.

▶ **Lemma 16.** *The problem whether there exists a non-decreasing SCC $\mathbb{B}$ of $\mathcal{A}_\sigma$ for some $\sigma \in \Sigma_{\mathrm{MD}}$ such that $\mathbb{B}$ contains a given state $p \in Q$ is* **NP***-complete.*

6 Conclusions

We introduced new estimates for measuring the asymptotic complexity of probabilistic programs and their VASS abstractions. We demonstrated the advantages of these measures over the asymptotic analysis of expected values, and we have also shown that tight complexity estimates can be computed efficiently for certain subclasses of VASS MDPs.

A natural continuation of our work is extending the results achieved for one-dimensional VASS MDPs to the multi-dimensional case. In particular, an interesting open question is whether the polynomial asymptotic analysis for non-deterministic VASS presented in [23] can be generalized to VASS MDPs. Since the study of multi-dimensional VASS MDPs is notoriously difficult, a good starting point would be a complete understanding of VASS MDPs with two counters.

References

1 M. Ajdarów and A. Kučera. Deciding polynomial termination complexity for VASS programs. In *Proceedings of CONCUR 2021*, volume 203 of *Leibniz International Proceedings in Informatics*, pages 30:1–30:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

2 M Ajdarów and A. Kučera. Asymptotic complexity estimates for probabilistic programs and their vass abstractions. *arXiv*, 2307.04707 [cs.FL], 2023.

3 T. Brázdil, K. Chatterjee, A. Kučera, P. Novotný, and D. Velan. Deciding fast termination for probabilistic VASS with nondeterminism. In *Proceedings of ATVA 2019*, volume 11781 of *Lecture Notes in Computer Science*, pages 462–478. Springer, 2019.

4 T. Brázdil, K. Chatterjee, A. Kučera, P. Novotný, D. Velan, and F. Zuleger. Efficient algorithms for asymptotic bounds on termination time in VASS. In *Proceedings of LICS 2018*, pages 185–194. ACM Press, 2018.

5 M. Broy and M. Wirsing. On the algebraic specification of nondeterministic programming languages. In *Proceedings of CAAP'81*, volume 112 of *Lecture Notes in Computer Science*, pages 162–179. Springer, 1981.

6 K. Chatterjee and L. Doyen. Energy parity games. In *Proceedings of ICALP 2010, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 599–610. Springer, 2010.

7 K. Chatterjee and L. Doyen. Energy and mean-payoff parity Markov decision processes. In *Proceedings of MFCS 2011*, volume 6907 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2011.

8 K. Chatterjee, M. Henzinger, S. Krinninger, and D. Nanongkai. Polynomial-time algorithms for energy games with special weight structures. In *Proceedings of ESA 2012*, volume 7501 of *Lecture Notes in Computer Science*, pages 301–312. Springer, 2012.

9 W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki. The reachability problem for Petri nets is not elementary. In *Proceedings of STOC 2019*, pages 24–33. ACM Press, 2019.

10 J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

11 J.E. Hopcroft and J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8:135–159, 1979.

12 P. Jančar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148(2):281–301, 1995.

13 M. Jurdziński, R. Lazić, and S. Schmitz. Fixed-dimensional energy games are in pseudo-polynomial time. In *Proceedings of ICALP 2015*, volume 9135 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2015.

14 A. Kučera. Algorithmic analysis of termination and counter complexity in vector addition systems with states: A survey of recent results. *ACM SIGLOG News*, 8(4):4–21, 2021.

15 A. Kučera, J. Leroux, and D. Velan. Efficient analysis of VASS termination complexity. In *Proceedings of LICS 2020*, pages 676–688. ACM Press, 2020.

16 J. Leroux. Polynomial vector addition systems with states. In *Proceedings of ICALP 2018*, volume 107 of *Leibniz International Proceedings in Informatics*, pages 134:1–134:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

17 J. Leroux and Ph. Schnoebelen. On functions weakly computable by Petri nets and vector addition systems. In *Reachability Problems*, volume 8762 of *Lecture Notes in Computer Science*, pages 190–202. Springer, 2014.

18 R. Lipton. The reachability problem requires exponential space. Technical report 62, Yale University, 1976.

19 E.W. Mayr and A.R. Meyer. The complexity of the finite containment problem for Petri nets. *Journal of the Association for Computing Machinery*, 28(3):561–576, 1981.

20 C.A. Petri. Kommunikation mit automaten. *Schriften des Institutes für Instrumentelle Mathematik*, 3, 1962.

21 M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proceedings of CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 745–761. Springer, 2013.

22 M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, 2017.

23 F. Zuleger. The polynomial complexity of vector addition systems with states. In *Proceedings of FoSSaCS 2020*, volume 12077 of *Lecture Notes in Computer Science*, pages 622–641. Springer, 2020.

Universal Quantification Makes Automatic Structures Hard to Decide

Christoph Haase
Department of Computer Science, University of Oxford, UK

Radosław Piórkowski
Department of Computer Science, University of Oxford, UK

Abstract

Automatic structures are structures whose universe and relations can be represented as regular languages. It follows from the standard closure properties of regular languages that the first-order theory of an automatic structure is decidable. While existential quantifiers can be eliminated in linear time by application of a homomorphism, universal quantifiers are commonly eliminated via the identity $\forall x \,.\, \Phi \equiv \neg(\exists x \,.\, \neg\Phi)$. If Φ is represented in the standard way as an NFA, a priori this approach results in a doubly exponential blow-up. However, the recent literature has shown that there are classes of automatic structures for which universal quantifiers can be eliminated by different means without this blow-up by treating them as first-class citizens and not resorting to double complementation. While existing lower bounds for some classes of automatic structures show that a singly exponential blow-up is unavoidable when eliminating a universal quantifier, it is not known whether there may be better approaches that avoid the naïve doubly exponential blow-up, perhaps at least in restricted settings.

In this paper, we answer this question negatively and show that there is a family of NFA representing automatic relations for which the minimal NFA recognising the language after eliminating a single universal quantifier is doubly exponential, and deciding whether this language is empty is ExpSpace-complete.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases automatic structures, universal projection, state complexity, tiling problems

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.13

Funding This work is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 852769, ARiAT).

Acknowledgements We would like to thank the anonymous reviewers for their comments and suggestions, which helped us to improve the presentation of this paper. We are grateful to Alessio Mansutti for spotting a bug in Section 4.

1 Introduction

Quantifier elimination is a standard technique to decide logical theories. A logical theory $\mathcal{T}$ admits quantifier elimination whenever for every quantifier free conjunction of literals $\Phi(x, y_1, \ldots, y_n)$ of $\mathcal{T}$ there is a quantifier free formula $\Psi(y_1, \ldots, y_n)$ such that $\mathcal{T} \models \exists x. \Phi \leftrightarrow \Psi$. Universal quantifiers can then be eliminated simply by applying the duality $\forall x \,.\, \Phi \equiv \neg(\exists x \,.\, \neg\Phi)$. If the formula Ψ above is effectively computable then $\mathcal{T}$ is decidable. For quantifier elimination procedures, the computationally most expensive step is the elimination of an existential quantifier, since negating a formula can be performed on a syntactic level.

Automatic structures [11, 12, 2] are a family of first-order structures whose corresponding first-order theory can be decided using automata-theoretic methods, as an alternative approach to syntactic quantifier elimination. In their simplest variant, automatic structures are relational first-order structures whose universe is isomorphic to a regular language

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 13; pp. 13:1–13:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$L \subseteq \Sigma^*$ over some alphabet Σ, and whose n-ary relations are interpreted as regular languages over $(\Sigma^n)^*$. It follows that the set of all satisfying assignments of a quantifier-free formula $\Phi(x_1, \ldots, x_{m+1})$ can be obtained as the language $\mathcal{L}(\mathcal{A}) \subseteq (\Sigma^{m+1})^*$ of some finite-state automaton $\mathcal{A}$. In this setting, eliminating existential quantifiers is easy. In order to obtain a finite-state automaton whose language encodes the satisfying assignments to $\exists x_{m+1} . \Phi$, it suffices to apply the homomorphism induced by the mapping $h\colon (\Sigma^{m+1}) \to (\Sigma^m)$ such that $h(u_1, \ldots, u_{m+1}) \coloneqq (u_1, \ldots, u_m)$ to $\mathcal{L}(\mathcal{A})$. This can be performed in linear time, even when $\mathcal{A}$ is non-deterministic. However, if $\mathcal{A}$ is non-deterministic then computing a finite-state automaton whose language encodes the complement of Φ is computationally difficult and may lead to an automaton with $2^{\Omega(|\mathcal{A}|)}$ many states. In particular, due to double complementation, eliminating a universal quantifier may *a priori* lead to an automaton with $2^{2^{\Omega(|\mathcal{A})|}}$ many states. Notable examples of automatic structures are Presburger arithmetic [17], the first-order theory of the structure $\langle \mathbb{N}, 0, 1, +, = \rangle$, and its extension Büchi arithmetic [6, 4, 5]. Tool suites such as Lash [1], Tapas [15] and Walnut [16] are based on the automata-theoretic approach and have successfully been used to decide challenging instances of Presburger arithmetic and Büchi arithmetic from various application domains. Those tools eliminate universal quantifiers via double complementation.

Yet another approach to deciding Presburger arithmetic is based on manipulating semi-linear sets [10, 8], which are generalisations of ultimately periodic sets to arbitrary tuples of integers in $\mathbb{N}^d$. They are similar to automata-based methods in terms of the computational difficulty of existential projection and complementation: the former is easy whereas the latter is difficult.

For certain classes of automatic structures, it is possible to avoid eliminating universal quantifiers via existential projection and negation. For example, it was shown in [7] that deciding sentences of quantified integer programming $\exists \bar{x}_1 \forall \bar{x}_2 \ldots \exists \bar{x}_n . A \cdot \bar{x} \geq \bar{b}$ is complete for the n-th level of the polynomial hierarchy. The upper bound was obtained by manipulating so-called hybrid linear sets, which characterise the sets of integer solutions of systems of linear equations $A \cdot \bar{x} \geq \bar{b}$. A key technique introduced in [7] is called *universal projection* and enables directly eliminating universal quantifiers instead of resorting to double complementation and existential projection. Given $S \subseteq \mathbb{N}^{d+k}$, the universal projection of S onto the first d coordinates is defined as

$$\pi_d^{\forall}(S) \coloneqq \left\{ \bar{u} \in \mathbb{N}^d \;\middle|\; (\bar{u}, \bar{v}) \in S \text{ for all } \bar{v} \in \mathbb{N}^k \right\}.$$

It is shown in [7] that if S is a hybrid linear set then $\pi_d^{\forall}(S)$ is a hybrid linear set that can be obtained as a finite intersection of existential projections of certain hybrid linear sets. Moreover, the growth of the constants in the description of the hybrid linear set is only polynomial. Neither syntactic quantifier elimination nor automata-based methods are powerful enough to derive those tight upper bounds for quantified integer programming.

Another example is a recent paper of Boigelot et al. [3] showing that, in an automata-theoretic approach for a fragment of Presburger arithmetic with uninterpreted predicates, a universal projection step can directly be carried out on the automata level without complementation and only results in a singly exponential blowup.

Those positive algorithmic and structural results are specific to Presburger arithmetic and leave open the option that it may be possible to establish analogous results for general automatic structures. The starting point of this paper is the question of whether, given a non-deterministic finite automaton $\mathcal{A}$ whose language $\mathcal{L}(\mathcal{A}) \subseteq (\Sigma^{d+k})^*$ encodes the set of solutions of some quantifier-free formula Φ, there is a more efficient way to eliminate a (block of) universally quantified variable(s) than to first complement $\mathcal{A}$, next to perform an existential projection step, and finally to complement the resulting automaton again, especially in the light of the results of [7, 8]. Such a method would have direct consequences for tools such as Walnut which perform the aforementioned sequence of operations in order to eliminate universal quantifiers. In particular, Walnut is not restricted to automata resulting from formulas of linear arithmetic and allows users to directly specify a finite-state automaton when desired.

For better or worse, however, as the main result of this paper, we show that deciding whether the universal projection $\pi_d^\forall(\mathcal{L}(\mathcal{A}))$ of some language regular language $\mathcal{L}(\mathcal{A}) \subseteq \left(\Sigma^{d+k}\right)^*$ is empty is complete for ExpSpace. In particular, the lower bound already holds for $d = k = 1$, meaning that, in general, even for fixed-variable fragments of automatic structures, there is no algorithmically more efficient way to eliminate a single universal quantifier than the naïve one. The challenging part is to show the ExpSpace lower bound, which requires an involved reduction from a tiling problem. This reduction also enables us to show that there is a family $\left(\mathcal{A}_n\right)_{n\in\mathbb{N}}$ of non-deterministic finite automata such that $|\mathcal{A}_n| = O\left(n^3\right)$ and the smallest non-deterministic finite automaton recognising the universal projection of $\mathcal{L}(\mathcal{A}_n)$ has $\Omega\left(2^{2^n}\right)$ many states.

2 Preliminaries

2.1 Regular languages and their compositions

For a word $w = a_1a_2\cdots a_n \in \Sigma^*$, we write $w[i]$ to denote its i-th letter a_i, and $w[i,j]$ to denote the infix $a_ia_{i+1}\cdots a_j$ $(i \le j)$. We write $|w|$ for the length of w. A *proper suffix* of w is any infix $w[i,n]$ for some $1 < i \le n$.

Regular expressions. A *regular expression* over the alphabet Σ is a term featuring Kleene star, concatenation and union operations, as well as $\emptyset$ and all symbols from Σ as constants:

$$\mathcal{E}, \mathcal{E}' ::\equiv \mathcal{E}^* \mid \mathcal{E}\cdot\mathcal{E}' \mid \mathcal{E}+\mathcal{E}' \mid \emptyset \mid a \text{ for every } a \in \Sigma$$

For notational convenience, we also use sets of symbols $A \subseteq \Sigma$ as constants, and a k-fold concatenation $\mathcal{E}^k$ for every $k \in \mathbb{N}$; we also drop the concatenation dot most of the time. The language $\mathcal{L}(\mathcal{E}) \subseteq \Sigma^*$ is defined by structural induction, by interpreting constants as $\mathcal{L}(\emptyset) \coloneqq \emptyset$ and $\mathcal{L}(a) \coloneqq \{a\}$, and using the standard semantics of the three operations. The class of languages definable by regular expressions is called *regular languages*. The size $|\mathcal{E}|$ of a regular expression $\mathcal{E}$ is defined recursively as 1 plus the sizes of its subexpressions. For $\rho\colon \Sigma \to \Gamma$ and a regular expression $\mathcal{E}$, $\rho(\mathcal{E})$ is a regular expression over Γ obtained through substituting every constant $a \in \Sigma$ appearing in $\mathcal{E}$ by $\rho(a)$.

Finite-state automata. Regular languages can also be represented by *non-deterministic finite-state automata* (NFA). Such an automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_{\mathrm{I}}, Q_{\mathrm{F}})$, where Q is a finite non-empty set of *states*, Σ is a finite *alphabet*, $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*, $Q_{\mathrm{I}} \subseteq Q$ is the set of *initial states*, and $Q_{\mathrm{F}} \subseteq Q$ is the set of *final states*. A triple $(p, a, q) \in Q \times \Sigma \times Q$ is called a *transition* and denoted as $p \xrightarrow{a} q$. A *run* of $\mathcal{A}$ from a state q_0 to a state q_n $(n \in \mathbb{N})$ on a word $w = a_1a_2\cdots a_n \in \Sigma^*$ is a finite sequence of transitions $\left(q_{i-1} \xrightarrow{a_i} q_i\right)_{1\le i\le n}$ such that $q_{i-1} \xrightarrow{a_i} q_i \in \delta$ for every i. A word $w \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if there exists a run of $\mathcal{A}$ from some $q_{\mathrm{I}} \in Q_{\mathrm{I}}$ to $q_{\mathrm{F}} \in Q_{\mathrm{F}}$ over w. The *language* of $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) \coloneqq \{w \in \Sigma^* \mid w \text{ is accepted by } \mathcal{A}\}$. We define the size of $\mathcal{A}$ as $|\mathcal{A}| \coloneqq |Q| + |Q|^2 \cdot |\Sigma|$. This definition only depends on Q and Σ and ensures that $|\mathcal{A}| \ge |Q| + |\delta| \cdot |\Sigma|$. Subsequently, we will implicitly apply the well-known fact that the number of states of an NFA accepting the complement of $\mathcal{L}(\mathcal{A})$ is bounded by $2^{|Q|}$.

Below we state, without proofs, a few folklore properties of NFA:

▶ **Fact 1** (NFA closed under language union). *For any* NFA $\mathcal{A}, \mathcal{B}$ *over* Γ*, there exists an* NFA $\mathcal{A} \oplus \mathcal{B}$ *of size* $O(|\mathcal{A}| + |\mathcal{B}|)$ *such that* $\mathcal{L}(\mathcal{A} \oplus \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$.

▶ **Fact 2** (NFA closed under inverse language homomorphisms). *For any* NFA $\mathcal{A}$ *and a homomorphic mapping* $\rho\colon \Sigma^* \to \Gamma^*$*, there exists an* NFA $\rho^{-1}(\mathcal{A})$ *of size* $O(|\mathcal{A}|)$ *such that* $\mathcal{L}\big(\rho^{-1}(\mathcal{A})\big) = \rho^{-1}(\mathcal{L}(\mathcal{A}))$.

▶ **Fact 3** (NFA closed under concatenation of languages). *For any* NFA $\mathcal{A}, \mathcal{B}$ *there exists an* NFA $\mathcal{A} \odot \mathcal{B}$ *of size* $O(|\mathcal{A}| + |\mathcal{B}|)$ *s.t.* $\mathcal{L}(\mathcal{A} \odot \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cdot \mathcal{L}(\mathcal{B}) := \{u \cdot v \mid u \in \mathcal{L}(\mathcal{A})$ *and* $v \in \mathcal{L}(\mathcal{B})\}$.

▶ **Fact 4** (translating regular expressions into NFA). *For any regular expression* $\mathcal{E}$*, there exists an* NFA $\mathcal{A}(\mathcal{E})$ *such that* $|\mathcal{A}(\mathcal{E})| = O(|\mathcal{E}|)$ *and* $\mathcal{L}(\mathcal{A}(\mathcal{E})) = \mathcal{L}(\mathcal{E})$ *(see [19]).*

Filters. A *filter* is an auxiliary term introduced to simplify the proofs in Section 3, allowing for a modular design of regular languages. Fix a finite alphabet Σ and let $\Phi := \{\top, \bot\}$. Define homomorphisms $\psi_{\text{in}}, \psi_{\text{out}}\colon (\Sigma \times \Phi)^* \to \Sigma^*$ by their actions on a single letter

$$\underset{\text{(output every symbol from }\Sigma)}{\psi_{\text{in}}(a,b) := a} \qquad\qquad \underset{\text{(output only symbols paired with }\top)}{\psi_{\text{out}}(a,\top) := a \qquad \psi_{\text{out}}(a,\bot) := \varepsilon\,.}$$

A filter over an alphabet Σ is any language $F \subseteq (\Sigma \times \Phi)^*$. It induces a binary *input-output relation* $\mathcal{R}(F) \subseteq \Sigma^* \times \Sigma^*$ between input words u and their subsequences v:

$$(u,v) \in \mathcal{R}(F) \quad \overset{\text{def}}{\iff} \quad u = \psi_{\text{in}}(w) \text{ and } v = \psi_{\text{out}}(w) \text{ for some } w \in F\,.$$

We define $F(u) := \{v \mid (u,v) \in \mathcal{R}(F)\}$ to be the set of all possible outputs of F on u.

Filtering regular expressions. A *filtering regular expression* $\mathcal{F}$ over alphabet Σ is any regular expression over $\Sigma \times \Phi$. We write $\mathcal{F}(w) := \mathcal{L}(\mathcal{F})(w)$. To simplify the notation, we only write the Σ component of the constants, and underline parts of the expression. A symbol a appearing in an underlined fragment represents a pair $(a, \top)$, and in a fragment which is not underlined a pair $(a, \bot)$. Intuitively, underlined portions correspond to parts of the words being output. We apply the same notational convention to words $w \in (\Sigma \times \Phi)^*$. Additionally, for $\rho\colon \Sigma \to \Gamma$, we abuse the notation and extend it to the naturally defined homomorphism of type $\Sigma \times \Phi \to \Gamma \times \Phi$, which just preserves the coordinate belonging to Φ.

▶ **Example 5.** Fix $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$. Consider a filtering regular expression $\mathcal{F}$ and a word w, both over $A \cup \{␣\}$:

$$\mathcal{F} := (\underline{A}\, A^*\, ␣)^* \underline{A}\, A^* \qquad\qquad w := \texttt{nondeterministic␣finite␣automaton}\,.$$

We have:

$$\begin{aligned} &\mathcal{F}(w) = \{\texttt{nfa}\}\,,\\ &\quad \mathcal{F} = \Big((A \times \{\top\}) \cdot (A \times \{\bot\})^* \cdot (␣, \bot)\Big)^* \cdot (A \times \{\top\}) \cdot (A \times \{\bot\})^*\,,\\ &\mathcal{L}(\mathcal{F}) \ni \underline{\texttt{n}}\texttt{ondeterministic␣}\underline{\texttt{f}}\texttt{inite␣}\underline{\texttt{a}}\texttt{utomaton}\,. \end{aligned}$$

▶ **Fact 6.** *For every filtering regular expression* $\mathcal{F}$ *and* w*,* $\mathcal{F}(w) = \mathcal{L}(\mathcal{A}(\mathcal{F}))(w)$.

2.2 Automatic relations

Let Σ be a finite alphabet such that $\# \notin \Sigma$. We denote by $\Sigma_\# := \Sigma \cup \{\#\}$. Let $w_1, \ldots, w_k \in \Sigma^*$ such that $w_i = a_{i,1}a_{i,2}\cdots a_{i,\ell_i}$, and $\ell := \max\{\ell_1, \ldots, \ell_k\}$. For all $1 \leq i \leq k$ and $\ell_i < j \leq \ell$, set $a_{i,j} := \#$. The *convolution* $w_1 \otimes w_2 \otimes \cdots \otimes w_k$ of $w_1, \ldots, w_k$ is defined as

$$w_1 \otimes w_2 \otimes \cdots \otimes w_k := \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{k,1} \end{bmatrix} \begin{bmatrix} a_{1,2} \\ \vdots \\ a_{k,2} \end{bmatrix} \cdots \begin{bmatrix} a_{1,\ell} \\ \vdots \\ a_{k,\ell} \end{bmatrix} \subseteq \left(\Sigma_\#^k\right)^* .$$

For $R \subseteq (\Sigma^*)^k$ and $L \subseteq \left(\Sigma_\#^k\right)^*$ define

$$\begin{aligned} \mathit{Rel2Lang}(R) &:= \{w_1 \otimes w_2 \otimes \cdots \otimes w_k \mid (w_1, w_2, \ldots, w_k) \in R\}, \\ \mathit{Lang2Rel}(L) &:= \{(w_1, w_2, \ldots, w_k) \mid w_1 \otimes w_2 \otimes \cdots \otimes w_k \in L\}. \end{aligned}$$

In this paper, we say that a relation $R \subseteq (\Sigma^*)^k$ is *automatic* whenever $\mathit{Rel2Lang}(R)$ is regular. In the sequel, we assume that $\mathit{Rel2Lang}(R)$ is given by some NFA $\mathcal{A}_R = (Q, \Sigma_\#^k, \delta, Q_\mathrm{I}, Q_\mathrm{F})$.

Clearly, not every NFA $\mathcal{A} = (Q, \Sigma_\#^k, \delta, Q_\mathrm{I}, Q_\mathrm{F})$ is associated with an automatic relation $R \subseteq \Sigma^k$ since there are *a priori* no restrictions on the occurrences of the padding symbol "#". The language $L_{\text{✗}} \subseteq (\Sigma_\#^k)^*$ of all incorrect words that cannot be obtained as a convolution of words $w_1, \ldots, w_k \in \Sigma^*$ can be characterized by the following regular expression:

$$\left(\Sigma_\#^k\right)^* \cdot \left(\{\#\}^k + \sum_{1\leq i\leq k} \left(\left(\Sigma_\#^{i-1} \times \{\#\} \times \Sigma_\#^{k-i}\right) \cdot \left(\Sigma_\#^{i-1} \times \Sigma \times \Sigma_\#^{k-i}\right)\right)\right) \cdot \left(\Sigma_\#^k\right)^* .$$

This regular expression "guesses" that either a letter consisting solely of k # symbols occurs, or in some row of a word in $\left(\Sigma_\#^k\right)^*$ a "#" symbol is followed by a symbol in Σ. The language of this regular expression can be implemented by an NFA with $k+2$ many states. Hence, the complement $L_{\text{✓}} := \overline{L_{\text{✗}}}$ of $L_{\text{✗}}$, characterizing all "good" words, can be recognized by an NFA with 2^{k+2} many states. For the sake of readability, we do not parameterize $L_{\text{✗}}$ explicitly with k; the relevant k will always be clear from the context.

The *(existential) projection* of $R \subseteq (\Sigma^*)^{d+k}$ onto the first d components is defined as

$$\pi_d^\exists(R) := \left\{\bar{u} \in (\Sigma^*)^d \;\middle|\; (\bar{u}, \bar{w}) \in R \text{ for some } \bar{w} \in (\Sigma^*)^k\right\}.$$

The dual of existential projection is *universal projection*:

$$\pi_d^\forall(R) := \left\{\bar{u} \in (\Sigma^*)^d \;\middle|\; (\bar{u}, \bar{w}) \in R \text{ for all } \bar{w} \in (\Sigma^*)^k\right\}.$$

It is clear that $\pi_d^\forall(R) = \overline{\pi_d^\exists(\overline{R})}$. We overload the projection notation for languages

$$\pi_d^\exists(L) := \mathit{Rel2Lang}\big(\pi_d^\exists(\mathit{Lang2Rel}(L))\big) \qquad \pi_d^\forall(L) := \mathit{Rel2Lang}\big(\pi_d^\forall(\mathit{Lang2Rel}(L))\big).$$

In this article, given $\mathcal{A}_R$ such that $\mathit{Rel2Lang}(R) = \mathcal{L}(\mathcal{A}_R) \subseteq \left(\Sigma_\#^{d+k}\right)^*$, we are concerned with the computational complexity of deciding whether $\pi_d^\forall(R) = \emptyset$, measured in terms of $|\mathcal{A}_R|$. In Sections 3 and 5 we will prove the following.

▶ **Theorem 7.** *Deciding whether $\pi_d^\forall(R) \neq \emptyset$ for an automatic relation $R \subseteq (\Sigma^*)^{d+k}$ with an associated* NFA *$\mathcal{A}_R$ is* ExpSpace*-complete. The lower bound already holds for $d = k = 1$.*

3 Emptiness after universal projection is ExpSpace-hard

3.1 Tiling problems

Let $\mathcal{T} \subseteq_{\text{fin}} \mathbb{N}^4$ be a set of *tiles* with colours coded as tuples of numbers in top–right–bottom–left order. We define natural projections $\mathit{top}, \mathit{right}, \mathit{bottom}, \mathit{left} \colon \mathbb{N}^4 \to \mathbb{N}$ to access individual colours of a tile, and let $\mathit{colours}(\mathcal{T}) := \mathit{top}(\mathcal{T}) \cup \mathit{right}(\mathcal{T}) \cup \mathit{bottom}(\mathcal{T}) \cup \mathit{left}(\mathcal{T})$.

▶ **Example 8** (a tile). A tile $t = (2, 4, 3, 3)$ is drawn as with various auxiliary background shades corresponding to colour values.

A $\mathcal{T}$-*tiling of size* $(h, w) \in \mathbb{N}_+^2$ is any $h \times w$ matrix $T = [t_{i,j}]_{i,j} \in \mathcal{T}^{h \times w}$. It is *valid*, whenever colours of the neighboring tiles match:

$$\mathit{bottom}(t_{i,j}) = \mathit{top}(t_{i+1,j}) \qquad \text{for every } 1 \le i \le h-1 \text{ and } 1 \le j \le w, \tag{1}$$

$$\mathit{right}(t_{i,j}) = \mathit{left}(t_{i,j+1}) \qquad \text{for every } 1 \le i \le h \quad \text{and } 1 \le j \le w-1. \tag{2}$$

See Figure 1 on page 14 for an example of a valid tiling. A $\mathcal{T}$-*tiling of width* $w \in \mathbb{N}_+$ is any tiling in $\mathcal{T}^{h \times w}$ for some $h \in \mathbb{N}_+$. We define

$$\mathcal{T}^{\star \times w} := \bigcup_{h \in \mathbb{N}_+} \mathcal{T}^{h \times w}.$$

Additionally, for two distinguished tiles $t_{\nearrow}, t_{\swarrow} \in \mathcal{T}$, let $(\mathcal{T}, t_{\nearrow}, t_{\swarrow})$-tiling be any $\mathcal{T}$-tiling with $t_{\nearrow}$ placed in its top-right corner, and $t_{\swarrow}$ in its bottom-left corner.

▶ **Problem 9.** *CorridorTiling*

Input: A 4-tuple $(\mathcal{T}, t_{\nearrow}, t_{\swarrow}, n)$, where
- $\mathcal{T} \subseteq_{\text{fin}} \mathbb{N}^4$ is a finite set of tiles,
- $t_{\nearrow}, t_{\swarrow} \in \mathcal{T}$,
- $n \in \mathbb{N}$ given in unary.

Question: Does there exist a valid $(\mathcal{T}, t_{\nearrow}, t_{\swarrow})$-tiling of width 2^n?

By $\mathbb{T} \subset \mathcal{P}_{\text{fin}}(\mathbb{N}^4) \times \mathbb{N}^4 \times \mathbb{N}^4 \times \mathbb{N}_+$ we denote the set of all valid instances of the above problem.

▶ **Fact 10.** *CorridorTiling (Problem 9) is ExpSpace-hard.*

It is part of the folklore of the theory of computation that tiling problems can simulate the computation of Turing machines, the width of the requested tiling corresponding to the length of tape the machine is allowed to use. ExpSpace-completeness of a variant similar to the one above is sketched in [18].

3.2 The reduction

We prove Theorem 7 by a reduction from *CorridorTiling*. We will show that the ExpSpace-hardness occurs in the simplest case of universal projection – projecting a binary relation to get a unary one. Intuitively, for each instance $\mathcal{I} = (\mathcal{T}, t_{\nearrow}, t_{\swarrow}, n)$ of *CorridorTiling*, we want to construct an automaton $\mathcal{A}_{\mathcal{I}}$ such that $\pi_1^{\forall}(\mathcal{L}(\mathcal{A}_{\mathcal{I}}))$ is not empty if, and only if, $\mathcal{I}$ is a YES-instance. Formally, we provide a family of LogSpace-constructible NFA $(\mathcal{A}_{\mathcal{I}})_{\mathcal{I} \in \mathbb{T}}$, each of size $O\big(n^3\big)$, over the alphabet $(\Sigma_{\mathcal{I}} \cup \{\#\})^2$ for some $\Sigma_{\mathcal{I}}$ and representing relation $\mathit{Lang2Rel}(\mathcal{L}(\mathcal{A}_{\mathcal{I}})) \subseteq (\Sigma_{\mathcal{I}}^*)^2$ such that

$$\pi_1^{\forall}(\mathcal{L}(\mathcal{A}_{\mathcal{I}})) \neq \emptyset \iff \text{there exists a valid } (\mathcal{T}, t_{\nearrow}, t_{\swarrow})\text{-tiling of width } 2^n. \tag{3}$$

For the rest of this section, we fix an instance $\mathcal{I} = (\mathcal{T}, t_{\nearrow}, t_{\swarrow}, n) \in \mathbb{T}$. Due to technical reasons, we assume that $n \geq 6$. Note that every instance $(\mathcal{T}, t_{\nearrow}, t_{\swarrow}, n)$ with $n < 6$ can be easily transformed into $(\mathcal{T}', t'_{\nearrow}, t'_{\swarrow}, 6)$, while preserving the (non)existence of a valid tiling.

In Section 3.3, we define $\Sigma_{\mathcal{I}}$, specify a language $L_{\mathcal{I}} \in \Sigma_{\mathcal{I}}^*$, and prove that:

▶ **Lemma 11.** *$L_{\mathcal{I}} \neq \emptyset \iff$ there exists a valid $(\mathcal{T}, t_{\nearrow}, t_{\swarrow})$-tiling of width 2^n.*

In turn in Section 3.4, we construct in LogSpace an NFA $\mathcal{A}_{\mathcal{I}}$ such that

▶ **Lemma 12.** *$\pi_1^{\forall}(\mathcal{L}(\mathcal{A}_{\mathcal{I}})) = L_{\mathcal{I}}$.*

This completes the proof of Theorem 7, the correctness of the reduction stemming directly from Lemmas 11 and 12.

3.3 Word encoding of tilings

Here, we provide $\Sigma_{\mathcal{I}}$ and an encoding $enc_{\mathcal{I}} \colon \mathcal{T}^{\star \times 2^n} \to \Sigma_{\mathcal{I}}^*$. Then we define $L_{\mathcal{I}}$ as an intersection of six conditions, and prove Lemma 11 by showing that it coincides with the language of encodings of valid tilings.

Let $N_n := \mathbb{N} \cap [0, n]$. Additionally, let $N_n^{\circledast k} := \{i \in N_n \mid i \circledast k\}$ for $\circledast \in \{<, =, >\}$ and $k \in \mathbb{N}$ (to be used in the next section). The alphabet $\Sigma_{\mathcal{I}}$ consists of three groups of symbols – tiles from $\mathcal{T}$, numbers from N_n, and auxiliary symbols:

$$\Sigma_{\mathcal{I}} := \mathcal{T} \cup N_n \cup \{\mathtt{A}, [\![,]\!], \langle, \rangle\}\,.$$

Above, the symbol A is a mnemonic – it marks places in Section 3.4 where we enforce “for-all”-type properties. In what follows, we print some symbols in colours (e.g., 3010 t 20103) to assist in understanding the construction – such designations are auxiliary and are not reflected in the alphabet. The encoding of runs makes use of the word $\textsc{Comb}_n \in N_n^*$

$$\textsc{Comb}_n := n\, \textsc{Comb}'_{n-1}\, n\,,$$

where the words $\left(\textsc{Comb}'_i\right)_{0 \leq i \leq n}$ are defined recursively as

$$\begin{aligned} \textsc{Comb}'_0 &:= 0 \\ \textsc{Comb}'_i &:= \textsc{Comb}'_{i-1}\, i\, \textsc{Comb}'_{i-1} && \text{for } 0 < i \leq n. \end{aligned}$$

Observe that $\textsc{Comb}_n$ has length exactly $2^n + 1$; that property is important in the upcoming construction.

▶ **Example 13.** $\textsc{Comb}_4$ is 40102010301020104 and has length 17.

We define the *encoding* function $enc_{\mathcal{I}} \colon \mathcal{T}^{\star \times 2^n} \to \Sigma_{\mathcal{I}}^*$ in three steps. Let $T = [t_{i,j}]_{i,j} \in \mathcal{T}^{h \times 2^n}$ for some $h \in \mathbb{N}$. The tile $t_{i,j}$ in T is represented as

$$encCell_{\mathcal{I}}(T, i, j) := \langle\, \textsc{Comb}_n[1, j]\, t_{i,j}\, \textsc{Comb}_n[j+1, 2^n+1]\, \mathtt{A} \,\rangle\,,$$

a single row is encoded as

$$encRow_{\mathcal{I}}(T, i) := [\![\prod_{1 \leq j \leq 2^n} encCell_{\mathcal{I}}(T, i, j)]\!]\,,$$

and finally, the encoding of the entire tiling is defined as

$$enc_{\mathcal{I}}(T) := \mathtt{A} \prod_{1 \leq i \leq h} encRow_{\mathcal{I}}(T, i)\,.$$

▶ **Example 14.** The tiling $T = [t_{i,j}]_{i,j}$ of size $(2, 2^4)$ is encoded as

$$\mathtt{A}\,\llbracket\langle 4\, t_{1,1}\, 0102010301020104\, \mathtt{A}\rangle \cdots \langle 40102\, t_{1,5}\, 01030\cdots 04\, \mathtt{A}\rangle \cdots \langle 4010201030102010\, t_{1,16}\, 4\, \mathtt{A}\rangle\rrbracket\, \cdot$$
$$\cdot\, \llbracket\langle 4\, t_{2,1}\, 0102010301020104\, \mathtt{A}\rangle \cdots \langle 40102\, t_{2,5}\, 01030\cdots 04\, \mathtt{A}\rangle \cdots \langle 4010201030102010\, t_{2,16}\, 4\, \mathtt{A}\rangle\rrbracket.$$

The word above is written in two lines to make the correspondence to tiling more apparent.

Languages of encodings

Define the language of encodings of valid tilings of width 2^n with $t_{\nearrow}, t_{\swarrow}$ in the correct corners

$$\textsc{ValidEnc}_{\mathcal{J}} := \big\{ \mathit{enc}_{\mathcal{J}}(T) \;\big|\; T \text{ is a valid } (\mathcal{T}, t_{\nearrow}, t_{\swarrow})\text{-tiling of width } 2^n \big\}.$$

In order to express the notion of an encoding of a valid tiling in a more tangible way, below we define languages $\textsc{Cond}^1_{\mathcal{J}}, \ldots, \textsc{Cond}^6_{\mathcal{J}}$, which – as we prove in Lemma 15 – jointly characterise encodings. The first three of them are easily definable with automata of size $O(n)$, the next two guarantee an appropriate width of the encoding, while the last one enforces in a nontrivial way that the vertical colour match.

▶ **Condition 1.** Language $\textsc{Cond}^1_{\mathcal{J}}$ is given by the regular expression

$$\mathcal{E}^1_{\mathcal{J}} := \Big(\llbracket \langle\, n\, \mathcal{T}\, N_n^*\, \mathtt{A} \rangle \Big(\langle\, N_n^*\, \mathcal{T}\, N_n^*\, \mathtt{A}\, \rangle \Big)^* \langle\, N_n^*\, \mathcal{T}\, n\, \mathtt{A} \rangle \rrbracket \Big)^* .$$

Intuitively, encodings consist of rows bounded by $\llbracket$ and $\rrbracket$; each row comprised of cells delimited by $\langle$ and $\rangle$; the first cell begins with the number n followed by a tile, while last one ends with a tile, n and $\mathtt{A}$. As $|\mathcal{E}^1_{\mathcal{J}}| = O(n)$, by Fact 4 the language $\textsc{Cond}^1_{\mathcal{J}}$ is recognised by an NFA $\mathcal{B}^1_{\mathcal{J}} := \mathcal{A}\big(\mathcal{E}^1_{\mathcal{J}}\big)$ of size $O(n)$.

▶ **Condition 2.** The language $\textsc{Cond}^2_{\mathcal{J}}$ is defined by the regular expression

$$\mathcal{E}^2_{\mathcal{J}} := \llbracket \Big(\langle N_n^*\, \mathcal{T}\, N_n^*\, \mathtt{A} \rangle \Big)^* \langle N_n^*\, t_{\nearrow}\, N_n^*\, \mathtt{A} \rangle \rrbracket\; \Sigma_{\mathcal{J}}^*\; \llbracket \langle N_n^*\, t_{\swarrow}\, N_n^*\, \mathtt{A} \rangle \Big(\langle N_n^*\, \mathcal{T}\, N_n^*\, \mathtt{A} \rangle \Big)^* \rrbracket\, .$$

Trivially, this requires the first row of a purported tiling to end with $t_{\nearrow}$, and the last row to begin with $t_{\swarrow}$. As in Condition 1, $\textsc{Cond}^2_{\mathcal{J}}$ is recognised by an NFA $\mathcal{B}^2_{\mathcal{J}} := \mathcal{A}\big(\mathcal{E}^2_{\mathcal{J}}\big)$ of size $O(n)$.

▶ **Condition 3.** Let $Q = \mathit{colours}(\mathcal{T})$ and $\mathcal{B}^3_{\mathcal{J}} = (Q, \Sigma_{\mathcal{J}}, \delta, Q, Q)$, where δ has transitions

$$\begin{array}{ll} i \xrightarrow{t} j & \text{for every } i, j \in Q \text{ and } t \in \mathcal{T} \text{ s.t. } \mathit{left}(t) = i \text{ and } \mathit{right}(t) = j, \\ i \xrightarrow{a} i & \text{for every } i \in Q \text{ and } a \in \Sigma_{\mathcal{J}} \setminus (\mathcal{T} \cup \{\rrbracket\}), \\ i \xrightarrow{\rrbracket} j & \text{for every } i, j \in Q. \end{array}$$

We set $\textsc{Cond}^3_{\mathcal{J}} := \mathcal{L}\big(\mathcal{B}^3_{\mathcal{J}}\big)$; it contains encodings where tile colours match horizontally.

▶ **Condition 4** (each cell contains a $\textsc{Comb}_n$). The definition of $\textsc{Cond}^4_{\mathcal{J}}$ uses a filtering regular expression $\mathcal{F}^4_{\mathcal{J}}$:

$$\mathcal{F}^4_{\mathcal{J}} := \langle\, \underline{N_n^*}\, \mathcal{T}\, \underline{N_n^*\, \mathtt{A}}\, \rangle\, \Sigma_{\mathcal{J}}^*$$
$$\textsc{Cond}^4_{\mathcal{J}} := \big\{ w \in \Sigma_{\mathcal{J}}^* \;\big|\; \textsc{Comb}_n\, \mathtt{A} \in \mathcal{F}^4_{\mathcal{J}}(v) \text{ for every proper suffix } v \text{ of } w \text{ s.t. } v[1] = \langle \big\}$$

▶ **Condition 5** (prefix of a cell and first symbols of following cells' suffixes form a $\textsc{Comb}_n$).

$$\mathcal{F}^5_{\mathcal{J}} := \langle\, \underline{N_n^*}\, \mathcal{T}\, \underline{N_n}\, N_n^*\, \mathtt{A}\, \rangle \Big(\langle\, N_n^*\, \mathcal{T}\, \underline{N_n}\, N_n^*\, \mathtt{A}\, \rangle \Big)^* \langle\, N_n^*\, \mathcal{T}\, \underline{N_n}\, N_n^*\, \underline{\mathtt{A}}\, \rangle \rrbracket\, \Sigma_{\mathcal{J}}^*$$
$$\textsc{Cond}^5_{\mathcal{J}} := \big\{ w \in \Sigma_{\mathcal{J}}^* \;\big|\; \textsc{Comb}_n\, \mathtt{A} \in \mathcal{F}^5_{\mathcal{J}}(v) \text{ for every proper suffix } v \text{ of } w \text{ s.t. } v[1] = \langle \big\}$$

▶ **Condition 6** (tile colours match vertically). Let $\blacktriangledown_t := \{t' \in \mathcal{T} \mid top(t') = bottom(t)\}$ be the set of tiles with the top colour matching to the bottom of a tile t. Define

$$\mathcal{F}^6_{\mathcal{J}} := \sum_{t\in\mathcal{T}} \Big(\qquad \langle\, \underline{N_n^*}\ \ t\ \ N_n^*\,\mathtt{A}\,\rangle\,(\langle\, N_n^*\,\mathcal{T}\,N_n^*\,\mathtt{A}\,\rangle)^*\,\rrbracket\, \cdot$$

$$\cdot\, \llbracket(\langle\, N_n^*\,\mathcal{T}\,N_n^*\,\mathtt{A}\,\rangle)^*\,\langle\, N_n^*\,\blacktriangledown_t\,\underline{N_n^*}\,\mathtt{A}\,\rangle\,(\langle\, N_n^*\,\mathcal{T}\,N_n^*\,\mathtt{A}\,\rangle)^*\,\rrbracket\,\Sigma_{\mathcal{J}}^*\Big)\,.$$

The expression above was typeset in two lines only to highlight the correspondence between cells in two consecutive rows. Define the language $\textsc{Cond}^6_{\mathcal{J}}$ as

$$\textsc{Cond}^6_{\mathcal{J}} := \Big\{ w \in \Sigma_{\mathcal{J}}^* \;\Big|\; \textsc{Comb}_n\,\mathtt{A} \in \mathcal{F}^6_{\mathcal{J}}(v) \text{ for every proper suffix } v \text{ of } w \text{ such that } \\ v[1] = \langle \text{ and } v[j] = \llbracket \text{ for some } j \Big\}$$

Intuitively, requiring $\llbracket$ to appear in v filters out suffixes of the last row.

Define $L_{\mathcal{J}} := \mathtt{A}\bigcap_{1\le i\le 6}\textsc{Cond}^i_{\mathcal{J}}$. To prove Lemma 11, it suffices to show the following:

▶ **Lemma 15.** $L_{\mathcal{J}} = \textsc{ValidEnc}_{\mathcal{J}}$

Proof. The inclusion $L_{\mathcal{J}} \supseteq \textsc{ValidEnc}_{\mathcal{J}}$ is trivial.

Inclusion $L_{\mathcal{J}} \subseteq \textsc{ValidEnc}_{\mathcal{J}}$. Take any $u \in L_{\mathcal{J}}$. Due to Condition 1, it has the form $\mathtt{A}\prod_{1\le i\le h}\llbracket\, v_i\,\rrbracket$, where each $v_i \in (\Sigma_{\mathcal{J}}\setminus\{\llbracket,\rrbracket\})^*$. We will show that $\llbracket\, v_i\,\rrbracket \in Range(encRow_{\mathcal{J}}(\cdot))$ for all i. Fix an arbitrary i. Again due to Condition 1, v_i has the form

$$\prod_{1\le j\le w_i} (\langle\, p_{i,j}\, t_{i,j}\, s_{i,j}\,\mathtt{A}\,\rangle)\,,$$

where $w_i \in \mathbb{N}$, $p_{i,j}, s_{i,j} \in N_n^*$, $p_{i,1} = s_{i,w_i} = n$, and $t_{i,j} \in \mathcal{T}$. Due to Condition 5, we have that all $s_{i,j}$ are nonempty and

$$p_{i,1}\, s_{i,1}[1]\, s_{i,2}[1]\, s_{i,3}[1] \cdots s_{i,w_i}[1] = \textsc{Comb}_n\,. \tag{4}$$

This implies that $w_i = 2^n$. By Condition 5 and Equation (4) we get that $p_{i,j} = \textsc{Comb}_n[1,j]$, and now Condition 4 implies that $s_{i,j} = \textsc{Comb}_n[j+1, 2^n+1]$, so $\llbracket\, v_i\,\rrbracket$ is a valid encoding of a row of length 2^n. Hence u encodes a tiling $T := [t_{i,j}]_{i,j} \in \mathcal{T}^{h\times 2^n}$. Property (2) in the definition of a valid tiling is now trivially implied by Condition 3, and we only need to show (1). Fix arbitrary pair of tiles $t_{i,j}, t_{i+1,j}$ which are vertical neighbours. Observe that $p_{i,j}s_{i+1,x} = \textsc{Comb}_n \iff x = j$. Therefore, by Condition 6, $bottom(t_{i,j}) = top(t_{i+1,j})$, thus T is a valid $\mathcal{T}$-tiling, and – by Condition 2 – a valid $(\mathcal{T}, t_{\nearrow}, t_{\swarrow})$-tiling. ◀

3.4 Construction of the automaton $\mathcal{A}_{\mathcal{J}}$

Let $\Sigma_{\mathcal{J},\#} := \Sigma_{\mathcal{J}} \cup \{\#\}$. Here, we define the NFA $\mathcal{A}_{\mathcal{J}}$ over $\Sigma^2_{\mathcal{J},\#}$ and prove Lemma 12, which states that $\pi_1^{\forall}(\mathcal{L}(\mathcal{A}_{\mathcal{J}})) = L_{\mathcal{J}}$. The construction we present in this section, however, does not require the full generality of the setting of automatic structures:

- $Lang2Rel(\mathcal{L}(\mathcal{A}_{\mathcal{J}}))$ only holds for words of the same length, i.e., $\mathcal{A}_{\mathcal{J}}$ rejects words with #;
- we only use a subset of the alphabet: $\Sigma_{\mathcal{J}} \times N_n \subseteq \Sigma^2_{\mathcal{J},\#}$.

For this reason, we begin with a simplifying Lemma 16, which allows us to focus only on words satisfying above properties. Let $\rho_{\mathcal{J}} \colon (\Sigma_{\mathcal{J}} \times N_n)^* \to \Sigma_{\mathcal{J}}^*$ be a homomorphism given by $\rho_{\mathcal{J}}(a,\cdot) := a$. Additionally, let

$$\rho^{\forall}_{\mathcal{J}}(L) := \big\{ w \in \Sigma_{\mathcal{J}}^* \;\big|\; \rho_{\mathcal{J}}^{-1}(w) \subseteq L \big\}\,.$$

▶ **Lemma 16** (simplification). *For any* NFA $\mathcal{A}'_{\mathcal{J}}$ *over* $\Sigma_{\mathcal{J}} \times N_n$*, there exists an* NFA $\mathcal{A}_{\mathcal{J}}$ *over* $\Sigma^2_{\mathcal{J},\texttt{\#}}$ *such that* $\pi_1^{\forall}(\mathcal{L}(\mathcal{A}_{\mathcal{J}})) = \rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{A}'_{\mathcal{J}}))$.

Proof. Take any $\mathcal{A}'_{\mathcal{J}}$ over $\Sigma_{\mathcal{J}} \times N_n$. Let

$$\begin{aligned}
\mathcal{E}_1 &\coloneqq \big(\Sigma^2_{\mathcal{J}}\big)^*(\Sigma_{\mathcal{J}} \times \{\texttt{\#}\})^+ + \big(\Sigma^2_{\mathcal{J}}\big)^*(\{\texttt{\#}\} \times \Sigma_{\mathcal{J}})^+ && (u \otimes v \text{ such that } |u| \neq |v|)\\
\mathcal{E}_2 &\coloneqq \big(\Sigma^2_{\mathcal{J}}\big)^*(\Sigma_{\mathcal{J}} \times (\Sigma_{\mathcal{J}} \setminus N_n))\big(\Sigma^2_{\mathcal{J}}\big)^* && (\text{words with letter from } \Sigma^2_{\mathcal{J}} \setminus \Sigma_{\mathcal{J}} \times \mathbb{N}_n)\\
\mathcal{A}_{\mathcal{J}} &\coloneqq \mathcal{A}'_{\mathcal{J}} \oplus \mathcal{A}(\mathcal{E}_1) \oplus \mathcal{A}(\mathcal{E}_2)\,.
\end{aligned}$$

By definition, a word w belongs to $\pi_1^{\forall}(\mathcal{L}(\mathcal{A}_{\mathcal{J}}))$ whenever for all v the word $w \otimes v$ belongs to $\mathcal{L}(\mathcal{A}_{\mathcal{J}})$. By construction, $\mathcal{L}(\mathcal{A}_{\mathcal{J}})$ contains all $w \otimes v$ where $|v| \neq |w|$ ($\mathcal{E}_1$) or where v is using a symbol from $\Sigma_{\mathcal{J}} \setminus N_n$ ($\mathcal{E}_2$). Hence, the only words which can be missing from $\mathcal{L}(\mathcal{A}_{\mathcal{J}})$ come from $\mathcal{L}(\mathcal{A}'_{\mathcal{J}})$. This implies that $\pi_1^{\forall}(\mathcal{L}(\mathcal{A}_{\mathcal{J}})) = \rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{A}'_{\mathcal{J}}))$. ◀

Therefore, we only have to provide $\mathcal{A}'_{\mathcal{J}}$ such that $\rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{A}'_{\mathcal{J}})) = L_{\mathcal{J}}$. The construction is modular, based on six NFA corresponding to Conditions 1–6:

▶ **Lemma 17** (modular design). *For any six* NFA $(\mathcal{C}^i_{\mathcal{J}})_{1\leq i\leq 6}$ *over* $\Sigma_{\mathcal{J}} \times N_n$*, there exists an* NFA $\mathcal{A}'_{\mathcal{J}}$ *of size* $O\big(\sum_{1\leq i\leq 6}|\mathcal{C}^i_{\mathcal{J}}|\big)$ *over* $\Sigma_{\mathcal{J}} \times N_n$ *such that*

$$\rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{A}'_{\mathcal{J}})) = \texttt{A} \bigcap_{1\leq i\leq 6} \rho_{\mathcal{J}}^{\forall}\big(\mathcal{L}\big(\mathcal{C}^i_{\mathcal{J}}\big)\big)\,.$$

Proof. Define

$$\begin{aligned}
\mathcal{H} &\coloneqq (\{\texttt{A}\} \times N_n \setminus \{\texttt{1}, \texttt{2}, \dots, \texttt{6}\})\,(\Sigma_{\mathcal{J}} \times N_n)^*\\
\mathcal{A}'_{\mathcal{J}} &\coloneqq \mathcal{A}((\texttt{A}, \texttt{1})) \odot \mathcal{C}^1_{\mathcal{J}} \quad\oplus\quad \mathcal{A}((\texttt{A}, \texttt{2})) \odot \mathcal{C}^2_{\mathcal{J}} \quad\oplus\quad \cdots \quad\oplus\quad \mathcal{A}((\texttt{A}, \texttt{6})) \odot \mathcal{C}^6_{\mathcal{J}} \quad\oplus\quad \mathcal{A}(\mathcal{H})\,.
\end{aligned}$$

Observe that

$$\begin{aligned}
\texttt{A}w \in \rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{A}'_{\mathcal{J}})) \iff \rho_{\mathcal{J}}^{-1}(\texttt{A}w) \subseteq \mathcal{L}(\mathcal{A}'_{\mathcal{J}}) \iff (\{\texttt{A}\} \times N_n)\,\rho_{\mathcal{J}}^{-1}(w) \subseteq \mathcal{L}(\mathcal{A}'_{\mathcal{J}}) \iff\\
\iff \forall i \in N_n\,.\,(\texttt{A}, i)\,\rho_{\mathcal{J}}^{-1}(w) \subseteq \mathcal{L}(\mathcal{A}'_{\mathcal{J}})\,,
\end{aligned}$$

but trivially

$$\begin{aligned}
\mathcal{L}\Big(\mathcal{A}((\texttt{A}, j)) \odot \mathcal{C}^j_{\mathcal{J}}\Big) \cap (\texttt{A}, i)\,\rho_{\mathcal{J}}^{-1}(w) &= \emptyset && \text{for any } i \neq j\\
\mathcal{L}(\mathcal{A}(\mathcal{H})) \cap (\texttt{A}, i)\,\rho_{\mathcal{J}}^{-1}(w) &= \emptyset && \text{for any } i.
\end{aligned}$$

Therefore, $\texttt{A}w \in \rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{A}'_{\mathcal{J}}))$ if, and only if, $\rho_{\mathcal{J}}^{-1}(w) \subseteq \rho_{\mathcal{J}}^{\forall}\big(\mathcal{L}\big(\mathcal{C}^i_{\mathcal{J}}\big)\big)$ for all i, as required. ◀

By definition of $L_{\mathcal{J}}$, it only remains to construct automata $\mathcal{C}^i_{\mathcal{J}}$ such that $\rho_{\mathcal{J}}^{\forall}\big(\mathcal{L}\big(\mathcal{C}^i_{\mathcal{J}}\big)\big) = \textsc{Cond}^i_{\mathcal{J}}$ for $1 \leq i \leq 6$. The construction is easy for Conditions 1–3:

$$\mathcal{C}^i_{\mathcal{J}} \coloneqq \rho_{\mathcal{J}}^{-1}\big(\mathcal{B}^i_{\mathcal{J}}\big) \qquad\qquad \text{for } i \in \{1, 2, 3\}$$

as $\rho_{\mathcal{J}}^{\forall}\big(\mathcal{L}\big(\rho_{\mathcal{J}}^{-1}(\mathcal{A})\big)\big) = \mathcal{L}(\mathcal{A})$ for any NFA $\mathcal{A}$. Observe that the remaining Conditions 4–6 all speak about “every proper suffix” satisfying some simple regular property. We handle that in a general way. For $L \subseteq (\Sigma_{\mathcal{J}} \times N_n)^*$, define

$$L_{\forall\text{suf}}(L) \coloneqq \big\{w \mid v \in \rho_{\mathcal{J}}^{\forall}(L) \text{ for all proper suffixes } v \text{ of } w\big\}$$

▶ **Lemma 18** (recognising “for all proper suffixes”). *For any* NFA $\mathcal{A}$ *over* $\Sigma_{\mathcal{J}} \times N_n$*, there exists an* NFA $\textsc{AllSuf}(\mathcal{A})$ *of size* $O(|\mathcal{A}|)$ *such that*

$$\rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\textsc{AllSuf}(\mathcal{A}))) = L_{\forall\text{suf}}(\mathcal{L}(\mathcal{A}))\,.$$

Proof. Fix any NFA $\mathcal{A} = (Q, \Sigma_{\mathcal{J}} \times N_n, \delta, Q_{\mathrm{I}}, Q_{\mathrm{F}})$. We define $\textit{AllSuf}(\mathcal{A})$ which guesses the suffix to verify

$$\textit{AllSuf}(\mathcal{A}) := (Q \cup \{s\}, \Sigma_{\mathcal{J}} \times N_n, \delta \cup \delta', \{s\}, Q_{\mathrm{F}} \cup \{s\})$$

for some fresh state $s \notin Q$, and δ' containing transitions $s \xrightarrow{(a,0)} s$ for $a \in \Sigma_{\mathcal{J}}$ and $s \xrightarrow{(a,n)} q$ for $a \in \Sigma_{\mathcal{J}}$, $n \in N_n^{>0}$, $q \in Q_{\mathrm{I}}$. Additionally, let τ be a homomorphism such that $\tau(a) := (a, 0)$.

Inclusion "$\subseteq$". Take any $w \in \rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\textit{AllSuf}(\mathcal{A})))$. Let v be any proper suffix of w. Take any $v' \in \rho_{\mathcal{J}}^{-1}(v)$. We need to show that $v' \in \mathcal{L}(\mathcal{A})$. The word w can be written as uav, for $|u| \geq 0$ and $|a| = 1$. Consider a word $w' = \tau(u)(a,1)v'$. By definition of $\rho_{\mathcal{J}}^{\forall}$, $w' \in \mathcal{L}(\textit{AllSuf}(\mathcal{A}))$. Let r be an accepting run of $\textit{AllSuf}(\mathcal{A})$ over w'. By construction, the run stays in state s while reading $\tau(u)$ and goes to some $q \in Q_{\mathrm{I}}$ upon reading $(a,1)$. Therefore, the remaining suffix of r is an accepting run of $\mathcal{A}$ over v'.

Inclusion "$\supseteq$". Fix $w \in L_{\forall\mathrm{suf}}(\mathcal{L}(\mathcal{A}))$. Take any $w' \in \rho_{\mathcal{J}}^{-1}(w)$. We will show that $w' \in \mathcal{L}(\textit{AllSuf}(\mathcal{A}))$. Let $u'(a,k)v' := w'$ be such that u' is the maximal prefix arising as $\tau(u)$ for some u (possibly empty). Note that $k \neq 0$. By assumption, $v' \in \mathcal{L}(\mathcal{A})$, so there exists an accepting run r_2 of $\mathcal{A}$ over v' starting in some $q \in Q_{\mathrm{I}}$. By construction, there exists a run r_1 from s to q over $u'(a,k)$ in $\textit{AllSuf}(\mathcal{A})$. Hence the run $r_1 r_2$ accepts w'. ◀

To handle conditions "beginning with $\langle$" and "containing $\llbracket$" appearing as antecedents of implications, we proceed in the vein of the equivalence $a \to b \equiv \neg a \vee b$. Let

$$\mathcal{G}_{\neg\langle} := (\Sigma_{\mathcal{J}} \setminus \{\langle\})\,\Sigma_{\mathcal{J}}^{*} \qquad\qquad \mathcal{G}_{\neg\llbracket} := (\Sigma_{\mathcal{J}} \setminus \{\llbracket\})^{*}\,.$$

▶ **Lemma 19.** *For $i \in \{4,5,6\}$, given* NFA $\hat{\mathcal{C}}_{\mathcal{J}}^{i}$ *satisfying* $\rho^{\forall}(\mathcal{L}(\hat{\mathcal{C}}_{\mathcal{J}}^{i})) = \{w \mid \textit{Comb}_n\,\mathtt{A} \in \mathcal{F}_{\mathcal{J}}^{i}(w)\}$*, one can construct $\mathcal{C}_{\mathcal{J}}^{i}$ of size $O(|\hat{\mathcal{C}}_{\mathcal{J}}^{i}|)$ such that $\rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{C}_{\mathcal{J}}^{i})) = \textit{Cond}_{\mathcal{J}}^{i}$.*

Proof. Fix $\hat{\mathcal{C}}_{\mathcal{J}}^{4}, \hat{\mathcal{C}}_{\mathcal{J}}^{5}, \hat{\mathcal{C}}_{\mathcal{J}}^{6}$ as in the statement of the lemma. We define $\mathcal{C}_{\mathcal{J}}^{i}$ as

$$\begin{aligned}
\mathcal{C}_{\mathcal{J}}^{4} &:= \textit{AllSuf}\left(\hat{\mathcal{C}}_{\mathcal{J}}^{4} \oplus \rho_{\mathcal{J}}^{-1}\left(\mathcal{A}(\mathcal{G}_{\neg\langle})\right)\right)\\
\mathcal{C}_{\mathcal{J}}^{5} &:= \textit{AllSuf}\left(\hat{\mathcal{C}}_{\mathcal{J}}^{5} \oplus \rho_{\mathcal{J}}^{-1}\left(\mathcal{A}(\mathcal{G}_{\neg\langle})\right)\right)\\
\mathcal{C}_{\mathcal{J}}^{6} &:= \textit{AllSuf}\left(\hat{\mathcal{C}}_{\mathcal{J}}^{6} \oplus \rho_{\mathcal{J}}^{-1}\left(\mathcal{A}(\mathcal{G}_{\neg\langle} + \mathcal{G}_{\neg\llbracket})\right)\right).
\end{aligned}$$

The above cases are similar; w.l.o.g. let us focus on $\mathcal{C}^4$. Observe that

$$\rho^{\forall}\left(\mathcal{L}\left(\hat{\mathcal{C}}_{\mathcal{J}}^{4} \oplus \rho_{\mathcal{J}}^{-1}\left(\mathcal{A}(\mathcal{G}_{\neg\langle})\right)\right)\right) = \rho^{\forall}\left(\mathcal{L}\left(\hat{\mathcal{C}}_{\mathcal{J}}^{4}\right)\right) \cup \mathcal{L}\left(\mathcal{G}_{\neg\langle}\right) = \left\{w \mid \textit{Comb}_n\,\mathtt{A} \in \mathcal{F}_{\mathcal{J}}^{i}(w)\right\} \cup \mathcal{L}\left(\mathcal{G}_{\neg\langle}\right),$$

which directly corresponds to Condition 4, as required. ◀

The essential element needed to define NFA $\hat{\mathcal{C}}_{\mathcal{J}}^{i}$ as in Lemma 19 is an NFA for the language $\{\textit{Comb}_n\mathtt{A}\}$. First, we define $\textit{Comb}_n$ as the intersection of languages of $n+1$ regular expressions, then show how that can be concisely represented by an automaton $\mathcal{C}_n$ of size $O(n^2)$ such that $\rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{C}_n)) = \{\textit{Comb}_n\mathtt{A}\}$.

▶ **Definition 20.** *We define $n+1$ regular expressions $\mathcal{E}_i$ over $\Sigma_{\mathcal{J}}$*

$$\begin{aligned}
\mathcal{E}_0 &:= N_n^{>1}\left(\mathtt{0}\,N_n^{>1}\right)^{*}\\
\mathcal{E}_i &:= N_n^{>i}\left(\left(N_n^{<i}\right)^{*} i \left(N_n^{<i}\right)^{*} N_n^{>i}\right)^{*} && \textit{for } 0 < i < n\\
\mathcal{E}_n &:= n\left(N_n^{<n}\right)^{*} n
\end{aligned}$$

▶ **Lemma 21.** $\{\textit{Comb}_n\} = \bigcap_{0 \leq i \leq n} \mathcal{L}(\mathcal{E}_i)$.

Proof. It is easy to prove the inclusion "$\subseteq$" by unravelling the definition of $\textsc{Comb}_n$.

Inclusion "$\supseteq$". Take any $w \in \bigcap_{1\le i\le n}\mathcal{L}(\mathcal{E}_i)$. We will show that $w = \textsc{Comb}_n$.

▷ Claim 22. For $0 \le k \le n-1$, we have $\bigcap_{1\le i\le k}\mathcal{L}(\mathcal{E}_i) = \mathcal{L}\big(N_n^{>k}\,\big(\textsc{Comb}'_k\,N_n^{>k}\big)\big)^*$

We prove the claim by induction. The base case is trivial. Fix a word

$$w \in \mathcal{L}\big(N_n^{>k}\,\big(\textsc{Comb}'_k\,N_n^{>k}\big)\big)^* \cap \mathcal{L}(\mathcal{E}_{k+1})\,.$$

It has the form $w = a_1\,\textsc{Comb}'_k\,a_2\,\textsc{Comb}'_k\,\cdots\,\textsc{Comb}'_k\,a_m$ for some $m \ge 2$ and $a_1, a_2, \dots, a_m \in N_n^{>k}$. But since $w \in \mathcal{L}(\mathcal{E}_{k+1})$, every other symbol a_i is equal $k+1$ and m is odd. Thus

$$w = a_1\,\underbrace{\textsc{Comb}'_k\,(k+1)\,\textsc{Comb}'_k}_{\textsc{Comb}'_{k+1}}\cdots\underbrace{\textsc{Comb}'_k\,(k+1)\,\textsc{Comb}'_k}_{\textsc{Comb}'_{k+1}}\,a_m$$

We conclude by noticing that $\mathcal{L}(\mathcal{E}_n)\cap\mathcal{L}\Big(N_n^{>(n-1)}\Big(\textsc{Comb}'_{n-1}\,N_n^{>(n-1)}\Big)\Big)^* = \{\textsc{Comb}_n\}$. ◀

Let us define

$$\mathcal{C}_n \coloneqq \rho_{\mathcal{J}}^{-1}(\mathcal{A}(\mathcal{E}_0)) \odot \mathcal{A}((\mathtt{A},\mathtt{0})) \oplus \rho_{\mathcal{J}}^{-1}(\mathcal{A}(\mathcal{E}_1)) \odot \mathcal{A}((\mathtt{A},\mathtt{1})) \oplus \cdots \oplus \rho_{\mathcal{J}}^{-1}(\mathcal{A}(\mathcal{E}_n)) \odot \mathcal{A}((\mathtt{A},n))\,.$$

▶ **Lemma 23.** $\rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{C}_n)) = \Big(\bigcap_{0\le i\le n}\mathcal{L}(\mathcal{E}_i)\Big)\,\mathtt{A}$.

Proof. Inclusion "$\subseteq$". Take any $w = u\mathtt{A} \in \rho_{\mathcal{J}}^{\forall}(\mathcal{L}(\mathcal{C}_n))$, and $i \in N_n$. We prove that $u \in \mathcal{L}(\mathcal{E}_i)$. By definition, $\rho_{\mathcal{J}}^{-1}(u\mathtt{A}) \subseteq \mathcal{L}(\mathcal{C}_n)$. Fix a homomorphism $\tau(a) = (a, 0)$. Note that $\tau(u)(\mathtt{A}, i) \in \mathcal{L}(\mathcal{C}_n)$. This can be accepted only by the $\rho_{\mathcal{J}}^{-1}(\mathcal{A}(\mathcal{E}_i)) \odot \mathcal{A}((\mathtt{A}, i))$ component, thus $u \in \mathcal{L}(\mathcal{A}(\mathcal{E}_i)) = \mathcal{L}(\mathcal{E}_i)$, as required.

Inclusion "$\supseteq$". Take any $w = u\mathtt{A} \in \big(\bigcap_{0\le i\le n}\mathcal{L}(\mathcal{E}_i)\big)\,\mathtt{A}$. Take any $u'(\mathtt{A}, i) \in \rho_{\mathcal{J}}^{-1}(u\mathtt{A})$. Since $u \in \mathcal{L}(\mathcal{E}_i)$, $u' \in \rho_{\mathcal{J}}^{-1}(\mathcal{L}(\mathcal{E}_i))$, and $u'(\mathtt{A}, i) \in \mathcal{L}\big(\rho_{\mathcal{J}}^{-1}(\mathcal{A}(\mathcal{E}_i)) \odot \mathcal{A}((\mathtt{A}, i))\big)$, as required. ◀

▶ **Definition 24** (NFA $\hat{\mathcal{C}}^i_{\mathcal{J}}$). *Fix $i \in \{4,5,6\}$, NFA $\mathcal{C}_n = (Q^{(1)}, \Sigma \times N_n, \delta^{(1)}, Q_I^{(1)}, Q_F^{(1)})$ (of size $O(n^2)$) and $\mathcal{A}(\mathcal{F}^i_{\mathcal{J}}) = (Q^{(2)}, \Sigma \times \Phi, \delta^{(2)}, Q_I^{(2)}, Q_F^{(2)})$ (of size $O(n)$).*

Define $\hat{\mathcal{C}}^i_{\mathcal{J}} \coloneqq (Q, \Sigma \times N_n, \delta, Q_I, Q_F)$ of size $O(n^3)$, where

$$Q \coloneqq Q^{(1)} \times Q^{(2)}, \qquad Q_I \coloneqq Q_I^{(1)} \times Q_I^{(2)}, \qquad Q_F \coloneqq Q_F^{(1)} \times Q_F^{(2)},$$

and the transition relation is

$$\delta \coloneqq \Big\{(p,q) \xrightarrow{(a,\alpha)} (r,s)\ \Big|\ q \xrightarrow{(a,\top)} s \in \delta^{(2)} \wedge p \xrightarrow{(a,\alpha)} r \in \delta^{(1)}\Big\} \cup$$
$$\Big\{(p,q) \xrightarrow{(a,\alpha)} (p,s)\ \Big|\ q \xrightarrow{(a,\bot)} s \in \delta^{(2)} \wedge p \in Q^{(1)}\Big\}\,.$$

Intuitively, $\hat{\mathcal{C}}^i_{\mathcal{J}}$ runs $\mathcal{C}_n$ over the fragments of the input which were underlined by $\mathcal{F}^i_{\mathcal{J}}$.

▶ **Fact 25.** *$w \in \mathcal{L}\big(\hat{\mathcal{C}}^i_{\mathcal{J}}\big)$ if, and only if, $\exists v \in \mathcal{L}\big(\rho_{\mathcal{J}}^{-1}\big(\mathcal{F}^i_{\mathcal{J}}\big)\big)\,.\,\psi_{\mathrm{in}}(v) = w \wedge \psi_{\mathrm{out}}(v) \in \mathcal{L}(\mathcal{C}_n)$.*

To finish the construction, we need to prove that

▶ **Lemma 26.** *For $i \in \{4,5,6\}$*

$$\rho^{\forall}\big(\mathcal{L}\big(\hat{\mathcal{C}}^i_{\mathcal{J}}\big)\big) = \big\{w\ \big|\ \textsc{Comb}_n\,\mathtt{A} \in \mathcal{F}^i_{\mathcal{J}}(w)\big\}\,.$$

As the proofs for $i \in \{4, 5, 6\}$ are analogous, we focus on the hardest one, and then only comment how it can be adapted for $i \in \{4, 5\}$.

Proof ($i = 6$).

A. Inclusion "$\subseteq$". Take any $w \in \rho^{\forall}\big(\mathcal{L}\big(\hat{\mathcal{C}}^6_{\mathcal{J}}\big)\big)$. Define

$$U \coloneqq \big\{u \in \mathcal{L}\big(\mathcal{F}^6_{\mathcal{J}}\big) \mid \psi_{\text{in}}(u) = w\big\}$$

Note that if $U = \emptyset$, then $\mathcal{F}^6_{\mathcal{J}}(w) = \emptyset$, so by Fact 25 $\mathcal{L}\big(\hat{\mathcal{C}}^6_{\mathcal{J}}\big) = \emptyset$, and $\rho^{\forall}\big(\mathcal{L}\big(\hat{\mathcal{C}}^6_{\mathcal{J}}\big)\big) = \emptyset$, a contradiction. Therefore, $U \neq \emptyset$, and $w \in \mathcal{L}\big(\psi_{\text{in}}\big(\mathcal{F}^6_{\mathcal{J}}\big)\big)$, so it has the form

$$\langle\, p\, t\, s\, \mathtt{A} \,\rangle\, \beta \,\rrbracket\, \llbracket\, \langle\, p_1\, t_1\, s_1\, \mathtt{A} \,\rangle \langle\, p_2\, t_2\, s_2\, \mathtt{A} \,\rangle \cdots \langle\, p_k\, t_k\, s_k\, \mathtt{A} \,\rangle \,\rrbracket\, \gamma$$

for some $k \in \mathbb{N}$, $p, p_i, s, s_i \in N_n^*$, $t, t_i \in \mathcal{T}$, $\gamma \in (\Sigma_{\mathcal{J}} \setminus \{\llbracket, \rrbracket\})^*$ and $\gamma \in \Sigma_{\mathcal{J}}^*$. Furthermore, $|U| = k$ and it contains the following underlined words $u_1, \ldots, u_k \in (\Sigma_{\mathcal{J}} \times \Phi)^*$:

$$\begin{aligned}
u_1 &= \langle\, \underline{p}\, t\, s\, \mathtt{A} \,\rangle\, \beta \,\rrbracket\, \llbracket\, \langle\, p_1\, t_1\, \underline{s_1\, \mathtt{A}} \,\rangle \langle\, p_2\, t_2\, s_2\, \mathtt{A} \,\rangle \cdots \langle\, p_k\, t_k\, s_k\, \mathtt{A} \,\rangle \,\rrbracket\, \gamma \\
u_2 &= \langle\, \underline{p}\, t\, s\, \mathtt{A} \,\rangle\, \beta \,\rrbracket\, \llbracket\, \langle\, p_1\, t_1\, s_1\, \mathtt{A} \,\rangle \langle\, p_2\, t_2\, \underline{s_2\, \mathtt{A}} \,\rangle \cdots \langle\, p_k\, t_k\, s_k\, \mathtt{A} \,\rangle \,\rrbracket\, \gamma \\
&\ \vdots \\
u_k &= \langle\, \underline{p}\, t\, s\, \mathtt{A} \,\rangle\, \beta \,\rrbracket\, \llbracket\, \langle\, p_1\, t_1\, s_1\, \mathtt{A} \,\rangle \langle\, p_2\, t_2\, s_2\, \mathtt{A} \,\rangle \cdots \langle\, p_k\, t_k\, \underline{s_k\, \mathtt{A}} \,\rangle \,\rrbracket\, \gamma
\end{aligned}$$

Consider two cases, depending on the validity of the following assertion

$$\exists u \in U \,.\, \psi_{\text{out}}\big(\rho_{\mathcal{J}}^{-1}(u)\big) \subseteq \mathcal{L}(\mathcal{C}_n)$$

Case A.1: such u exists. Take any such $u \in U$. Observe that $\psi_{\text{out}}(u) \in \rho^{\forall}_{\mathcal{J}}(\mathcal{L}(\mathcal{C}_n)) = \{\textsc{Comb}_n\mathtt{A}\}$. Hence, $\psi_{\text{in}}(u) = w$, $\psi_{\text{out}}(u) = \textsc{Comb}_n\mathtt{A}$, and $u \in \mathcal{L}\big(\mathcal{F}^6_{\mathcal{J}}\big)$. Therefore, $\textsc{Comb}_n\mathtt{A} \in \mathcal{F}^6_{\mathcal{J}}(w)$, as required.

Case A.2: such u does not exist. Therefore, for every $u \in U$, there is some $v_u \in \rho_{\mathcal{J}}^{-1}(u)$ such that $\psi_{\text{out}}(v_u) \notin \mathcal{L}(\mathcal{C}_n)$. Fix any family $(v_u)_{u \in U}$ of such words. Let α_u be the position of the last underlined symbol in u. Fix a word $w' \in \rho_{\mathcal{J}}^{-1}(w)$ such that

$$w'[i] = \begin{cases} \psi_{\text{out}}(v_u[i]) & \text{if } i = \alpha_u \text{ for some } u \\ (w[i], 0) & \text{otherwise} \end{cases} .$$

Observe that w' is properly defined, as positions α_u are pairwise different (corresponding to the last letters of $s_1, s_2, \ldots, s_k$). Since $\rho_{\mathcal{J}}(w') = w$, from assumption $w \in \rho^{\forall}\big(\mathcal{L}\big(\hat{\mathcal{C}}^6_{\mathcal{J}}\big)\big)$ we have that $w' \in \mathcal{L}\big(\hat{\mathcal{C}}^6_{\mathcal{J}}\big)$. By Fact 25, we obtain $v \in \mathcal{L}\big(\rho_{\mathcal{J}}^{-1}\big(\mathcal{F}^6_{\mathcal{J}}\big)\big)$ such that

$$\psi_{\text{in}}(v) = w' \wedge \psi_{\text{out}}(v) \in \mathcal{L}(\mathcal{C}_n)$$

However, $\rho_{\mathcal{J}}(\psi_{\text{out}}(v)) = \rho_{\mathcal{J}}(\psi_{\text{out}}(v_u))$ for some $u \in U$ and last symbols of $\psi_{\text{out}}(v)$ and $\psi_{\text{out}}(v_u)$ are identical. Since by construction $\mathcal{C}_n$ ignores the component N_n of its alphabet $\Sigma_I \times N_n$ for all letters but the last one, we get that

$$\psi_{\text{out}}(v) \in \mathcal{L}(\mathcal{C}_n) \iff \psi_{\text{out}}(v_u) \in \mathcal{L}(\mathcal{C}_n)\,.$$

We conclude that $\psi_{\text{out}}(v) \notin \mathcal{L}(\mathcal{C}_n)$, a contradiction.

B. Inclusion "$\supseteq$". Take any w such that $\textsc{Comb}_n\mathtt{A} \in \mathcal{F}^6_{\mathcal{J}}(w)$. Using definition of $\mathcal{F}^6_{\mathcal{J}}(w)$, fix $v \in \mathcal{L}\big(\mathcal{F}^6_{\mathcal{J}}\big)$ such that $\psi_{\text{in}}(v) = w$ and $\psi_{\text{out}}(v) = \textsc{Comb}_n\mathtt{A}$. We have to show $\rho_{\mathcal{J}}^{-1}(w) \subseteq \mathcal{L}\big(\hat{\mathcal{C}}^6_{\mathcal{J}}\big)$. Take any $w' \in \rho_{\mathcal{J}}^{-1}(w)$. Let $u \in (\Sigma_{\mathcal{J}} \times \mathbb{N}_n \times \Phi)^*$ be the unique word such that $\psi_{\text{in}}(u) = w'$ and $\rho_{\mathcal{J}}(u) = v$. Observe that $\psi_{\text{out}}(w') \in \rho_{\mathcal{J}}^{-1}(\psi_{\text{out}}(w')) \subseteq \mathcal{L}(\mathcal{C}_n)$, thus $w' \in \mathcal{L}\big(\hat{\mathcal{C}}^6_{\mathcal{J}}\big)$, as required. ◀

Proof ($i \in \{4, 5\}$). The proof is analogous to the case $i = 6$. As the cases are distinguished by the filter $\mathcal{F}^i_{\mathcal{J}}$ being used, the only differences are related to the shape of words matched by $\psi_{\text{in}}(\mathcal{F}^i_{\mathcal{J}})$. In particular, the set U for $i \in \{4, 5\}$ is now a singleton containing u_i:

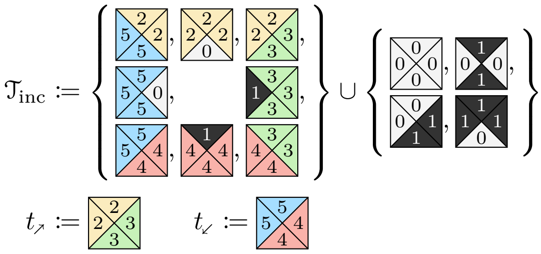
$$u_4 = \langle\, \underline{p}\, t\, \underline{s\, \mathtt{A}} \,\rangle\, \gamma \qquad (i = 4)$$

$$u_5 = \langle\, \underline{p_1}\, t\, \underline{s'_1}\, s_1\, \mathtt{A} \,\rangle \langle\, p_2\, t\, \underline{s'_2}\, s_2\, \mathtt{A} \,\rangle \cdots \langle\, p_{k-1}\, t\, \underline{s'_{k-1}}\, s_{k-1}\, \mathtt{A} \,\rangle \langle\, p_k\, t\, \underline{s'_k\, \mathtt{A}} \,\rangle]\!]\, \gamma \qquad (i = 5)$$

The rest of the proof only requires substituting $\mathcal{F}^6_{\mathcal{J}}$ with $\mathcal{F}^4_{\mathcal{J}}$ or $\mathcal{F}^5_{\mathcal{J}}$. ◀

4 NFA of doubly exponential size after universal projection

From the lower bounds established in Section 3.4, it is now easy to construct a family $\left(\mathcal{A}_{(\mathcal{T}_{\text{inc}}, t_\nearrow, t_\swarrow, n)}\right)_{n \in \mathbb{N}}$ of NFA, each of size $O(n^3)$, such that the smallest NFA after a universal projection step has doubly-exponentially many states. Indeed, let



Intuitively, the colours $0, 1$ vertically represent the counter bits, and horizontally encode the carryover bit. The only valid $(\mathcal{T}_{\text{inc}}, t_\nearrow, t_\swarrow)$-tiling of width n simulates incrementing an $(n - 2)$-bit binary counter from 0 to $2^{(n-2)} - 1$; see Figure 1 for an example with $n = 5$. Thus, after a universal projection step, the resulting NFA accepts a single word of length doubly exponential in n.

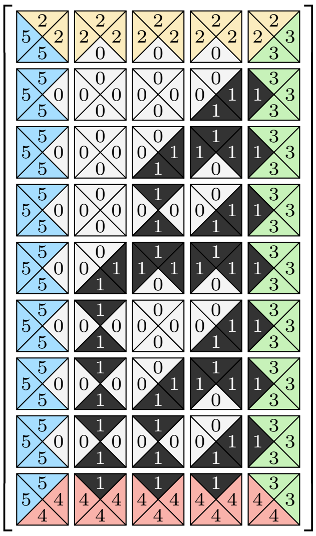


Figure 1 The unique valid $(\mathcal{T}_{\text{inc}}, t_\nearrow, t_\swarrow)$-tiling of width 5.

▶ **Proposition 27.** *The* NFA *for* $\pi^\forall_1\left(\mathcal{L}\left(\mathcal{A}_{(\mathcal{T}_{\text{inc}}, t_\nearrow, t_\swarrow, n)}\right)\right)$ *has size* $\Omega\left(2^{2^n}\right)$.

5 Emptiness after universal projection is in ExpSpace

We now consider algorithmic upper bounds for deciding whether the language of an automatic relation $R \subseteq (\Sigma^*)^{d+k}$ after a universal projection step is non-empty, measured in terms of the size of the associated NFA $\mathcal{A}_R$, which yields the upper bound of Theorem 7.

Define a homomorphism $h\colon (\Sigma_{\#}^{d+k})^* \to (\Sigma_{\#}^{d})^*$ by

$$h(a_1, \ldots, a_d, a_{d+1}, \ldots, a_{d+k}) \coloneqq (a_1, \ldots, a_d).$$

Given an NFA $\mathcal{B}$ over $\Sigma_{\#}^{d+k}$ such that $S \subseteq (\Sigma^*)^{d+k}$ is automatic via $\mathcal{B}$, it is clear that we can compute in linear time an NFA $\mathcal{B}'$ with the same number of states as $\mathcal{B}$ such that $L(\mathcal{B}') = h(\mathcal{L}(\mathcal{B}))$. The homomorphism h acts almost like existential projection, but in general, we do not have that $\pi_d^{\exists}(S)$ is automatic via $\mathcal{B}'$. For instance, suppose that

$$w = \begin{bmatrix} a \\ a \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} \begin{bmatrix} \# \\ c \end{bmatrix} \begin{bmatrix} \# \\ a \end{bmatrix} \in \mathcal{L}(\mathcal{B})\,.$$

Then $h(w) = aa\#\# \notin L_{\checkmark}$ because of the trailing # symbols. To remove them, we define

$$\textsc{Strip}(L) \coloneqq \left\{ w \;\middle|\; \text{there exists } v \in (\{\#\}^d)^* \text{ such that } wv \in L \right\}.$$

It is then the case that $\pi_d^{\exists}(S)$ is automatic via $\textsc{Strip}(\mathcal{L}(\mathcal{B}')) \cap L_{\checkmark}$. Note that an NFA for $\textsc{Strip}(L)$ can be computed in linear time from an NFA for L without changing the set of states by making all states accepting that can reach a final state via a sequence of "$\{\#\}^d$" symbols.

Recall that $\pi_d^{\forall}(R) = \overline{\pi_d^{\exists}(\overline{R})}$, consequently an automatic presentation of $\pi_d^{\forall}(R)$ is given by

$$\overline{\left(\textsc{Strip}\left(h\left(\overline{\mathcal{L}(\mathcal{A}_R)}\right)\right) \cap L_{\checkmark}\right)} \cap L_{\checkmark}.$$

Assuming Q is the set of states of $\mathcal{A}_R$, and recalling that $L_{\checkmark} \subseteq (\Sigma_{\#}^{d})^*$ is given by an NFA with 2^{d+2} many states, it can easily be checked that the number of states of an NFA whose language gives the universal projection of R is bounded by $2^{\left(2^{|Q|+d+2}\right)+d+2}$.

With those characterisations and estimations at hand, the ExpSpace upper bound stated in Theorem 7 can now easily be established.

▶ **Proposition 28.** *Deciding whether $\pi_d^{\forall}(R) \neq \emptyset$ is in* ExpSpace*, measured in terms of the size of its associated* NFA $\mathcal{A}_R$.

Proof. For an ExpSpace algorithm, we first construct an NFA $\mathcal{B} = (Q, \Sigma_{\#}^{d}, \delta, q_0, F)$ whose language is $\left(\textsc{Strip}\left(p_d(\overline{\mathcal{L}(\mathcal{A}_R)})\right) \cap L_{\checkmark}\right)$. We have $|Q| \leq 2^{|Q_R|+d+2}$, where Q_R is the set of states of $\mathcal{A}_R$, and hence $\mathcal{B}$ can be constructed in exponential space. It remains to show that non-emptiness of $\overline{\mathcal{L}(\mathcal{B})} \cap L_{\checkmark}$ can be decided in exponential space.

Clearly, we cannot explicitly construct an NFA for this language. Let $\mathcal{A}_{\checkmark} = (S, \Sigma_{\#}^{d}, \delta_{\checkmark}, s_0, F_{\checkmark})$ be the the NFA for $L_{\checkmark}$, we can however non-deterministically guess a word in $\overline{\mathcal{L}(\mathcal{B})} \cap \mathcal{L}(\mathcal{A}_{\checkmark})$ letter by letter as follows. We keep track of a configuration of the form $(Q', s) \in 2^Q \times S$, which initially is $(\{q_0\}, s_0)$. Then we repeatedly non-deterministically guess some $a \in \Sigma_{\#}^{d}$ and update (Q', s) to $(\delta(Q', a), \delta_{\checkmark}(s, a))$ until we reach a configuration (Q', s) such that $Q' \cap F = \emptyset$ and $s \in F_{\checkmark}$. Clearly, the word obtained by this sequence of letters is in $\overline{\mathcal{L}(\mathcal{B})}$ and $\mathcal{L}(L_{\checkmark})$. The overall membership in ExpSpace is then a consequence of Savitch's theorem and the observation that the length of the shortest word in $\overline{\mathcal{L}(\mathcal{B})} \cap L_{\checkmark}$ is bounded by $2^{\left(2^{|Q|+d+2}\right)+d+2}$. ◀

6 Conclusion

In this paper, we studied the computational complexity of eliminating universal quantifiers in automatic structures. We showed that, in general, this is a computationally challenging problem whose associated decision problem is ExpSpace-complete. Our result further reinforces the intuition already stemming from [13] that, in general, the alternation of quantifiers requires "complex" automata.

It would be interesting to understand whether it is possible to identify natural sufficient conditions on regular languages for which a universal projection step does not result in a doubly-exponential blow-up and only leads to, e.g., polynomial or singly exponential growth. Results of this kind have been obtained in model-theoretic terms for structures of bounded degree [14, 9], but we are not aware of a systematic study of questions of this kind on the level of regular languages.

References

1 The LASH toolset. `https://people.montefiore.uliege.be/boigelot/research/lash/index.html`.

2 Achim Blumensath and Erich Grädel. Automatic structures. In *Logic in Computer Science, LICS*, pages 51–62. IEEE Computer Society, 2000. `doi:10.1109/LICS.2000.855755`.

3 Bernard Boigelot, Pascal Fontaine, and Baptiste Vergain. First-order quantification over automata. In *Conference on Implementation and Application of Automata, CIAA*, Lect. Notes Comp. Sci. Springer, 2023. To appear. `doi:10.48550/arXiv.2306.04210`.

4 Véronique Bruyère. Entiers et automates finis. *Mémoire de fin d'études*, 1985.

5 Véronique Bruyère, Georges Hansel, Christian Michaux, and Roger Villemaire. Logic and p-recognizable sets of integers. *Bull. Belg. Math. Soc. Simon Stevin*, 1(2):191–238, 1994. `doi:10.36045/bbms/1103408547`.

6 J. Richard Büchi. Weak second-order arithmetic and finite automata. *Math. Logic Quart.*, 6(1-6):66–92, 1960. `doi:10.1002/malq.19600060105`.

7 Dmitry Chistikov and Christoph Haase. On the complexity of quantified integer programming. In *International Colloquium on Automata, Languages, and Programming, ICALP*, volume 80 of *LIPIcs*, pages 94:1–94:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ICALP.2017.94`.

8 Dmitry Chistikov, Christoph Haase, and Alessio Mansutti. Geometric decision procedures and the VC dimension of linear arithmetic theories. In *Logic in Computer Science, LICS*, pages 59:1–59:13. ACM, 2022. `doi:10.1145/3531130.3533372`.

9 Antoine Durand-Gasselin and Peter Habermehl. Ehrenfeucht-fraïssé goes elementarily automatic for structures of bounded degree. In *Symposium on Theoretical Aspects of Computer Science, STACS*, volume 14 of *LIPIcs*, pages 242–253. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. `doi:10.4230/LIPIcs.STACS.2012.242`.

10 Seymour Ginsburg and Edwin H. Spanier. Semigroups, Presburger formulas, and languages. *Pac. J. Math.*, 16(2):285–296, 1966.

11 Bernard R. Hodgson. On direct products of automaton decidable theories. *Theor. Comput. Sci.*, 19(3):331–335, 1982. `doi:10.1016/0304-3975(82)90042-1`.

12 Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. In *Logical and Computational Complexity, LCC*, volume 960 of *Lect. Notes Comp. Sci.*, pages 367–392. Springer, 1995. `doi:10.1007/3-540-60178-3_93`.

13 Dietrich Kuske. Theories of automatic structures and their complexity. In *Conference on Algebraic Informatics, CAI*, volume 5725 of *Lect. Notes Comp. Sci.*, pages 81–98. Springer, 2009. `doi:10.1007/978-3-642-03564-7_5`.

14 Dietrich Kuske and Markus Lohrey. Automatic structures of bounded degree revisited. *J. Symb. Log.*, 76(4):1352–1380, 2011. `doi:10.2178/jsl/1318338854`.

15 Jérôme Leroux and Gérald Point. TaPAS: The Talence Presburger Arithmetic Suite. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 5505 of *Lect. Notes Comp. Sci.*, pages 182–185. Springer, 2009. `doi:10.1007/978-3-642-00768-2_18`.

16 Hamoon Mousavi. Automatic theorem proving in Walnut. *CoRR*, 2016. `arXiv:1603.06017`.

17 Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I congres de Mathematiciens des Pays Slaves*, pages 92–101. 1929.

18 François Schwarzentruber. The complexity of tiling problems, 2019. `arXiv:1907.00102`.

19 Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968. `doi:10.1145/363347.363387`.

About Decisiveness of Dynamic Probabilistic Models

Alain Finkel
Université Paris-Saclay, CNRS, ENS Paris-Saclay, IUF,
Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

Serge Haddad
Université Paris-Saclay, CNRS, ENS Paris-Saclay,
Laboratoire Méthodes Formelles, INRIA 91190, Gif-sur-Yvette, France

Lina Ye
Université Paris-Saclay, CNRS, ENS Paris-Saclay, CentraleSupélec,
Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

Abstract

Decisiveness of infinite Markov chains with respect to some (finite or infinite) target set of states is a key property that allows to compute the reachability probability of this set up to an arbitrary precision. Most of the existing works assume constant weights for defining the probability of a transition in the considered models. However numerous probabilistic modelings require the (dynamic) weight to also depend on the current state. So we introduce a dynamic probabilistic version of counter machine (pCM). After establishing that decisiveness is undecidable for pCMs even with constant weights, we study the decidability of decisiveness for subclasses of pCM. We show that, without restrictions on dynamic weights, decisiveness is undecidable with a single state and single counter pCM. On the contrary with polynomial weights, decisiveness becomes decidable for single counter pCMs under mild conditions. Then we show that decisiveness of probabilistic Petri nets (pPNs) with polynomial weights is undecidable even when the target set is upward-closed unlike the case of constant weights. Finally we prove that the standard subclass of pPNs with a regular language is decisive with respect to a finite set whatever the kind of weights.

2012 ACM Subject Classification Mathematics of computing → Markov processes; Theory of computation → Concurrency

Keywords and phrases infinite Markov chain, reachability probability, decisiveness

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.14

Related Version *Full Version*: `https://arxiv.org/abs/2305.19564` [17]

Funding *Alain Finkel*: His work has been supported by ANR project BRAVAS (ANR-17-CE40-0028).
Serge Haddad: His work has been supported by ANR project MAVeriQ (ANR-20-CE25-0012).

1 Introduction

Infinite Markov chains. Since the 1980's, finite-state Markov chains have been considered for the modeling and analysis of probabilistic concurrent finite-state programs [27]. More recently this approach has been extended to the verification of the infinite-state Markov chains obtained from probabilistic versions of automata extended with unbounded data (like stacks, channels, counters, clocks). The problem of *Computing the Reachability Probability up to an arbitrary precision* (CRP) is a central problem in quantitative verification and it has been studied by many authors [23, 16, 3, 12].

Computing the probability of reachability. There are (at least) two strategies to solve the CRP problem.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 14; pp. 14:1–14:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The first one is to consider the Markov chains associated with a particular class of probabilistic models like probabilistic pushdown automata (pPDA) or probabilistic Petri nets (pPN) and some specific target sets and to exploit the properties of these models to design a CRP-algorithm. For instance in [12], the authors exhibit a PSPACE algorithm for pPDA and PTIME algorithms for single-state pPDA and for one-counter automata.

The second one consists in exhibiting a property of Markov chains that yields a generic algorithm for solving the CRP problem and then looking for models that generate Markov chains that fulfill this property. *Decisiveness* of Markov chains is such a property (a Markov chain is decisive w.r.t. a target if almost surely a random path either reaches the target or the target becomes unreachable) and it has been shown that pLCS are decisive and that probabilistic Petri nets (pPN) are decisive when the target set is upward-closed [3].

Two limits of the previous approaches. In most of the works, the probabilistic models associate a constant (also called *static*) weight for transitions and get transition probabilities by normalizing these weights among the enabled transitions in the current state (except for some semantics of pLCS like in [19] where transition probabilities depend on the state due to the possibility of message losses). This forbids to model phenomena like congestion in networks (resp. performance collapsing in distributed systems) when the number of messages (resp. processes) exceeds some threshold leading to an increasing probability of message arrivals (resp. process creations) before message departures (resp. process terminations). In order to handle them, one needs to consider *dynamic* weights i.e., weights depending on the current state.

Dynamic weights. The usual formalism for performance evaluation is the model of continuous time Markov chain (CTMC) (see for instance the book "Continuous-Time Markov Chains: An Applications-Oriented Approach". William J. Anderson). In this model, the transitions are labelled by a rate (of a negative exponential distribution). The underlying discrete time model (which is enough to study some important properties) is a DTMC obtained by normalizing the rates viewed as weights. To emphasize the relevance of dynamic and more specifically polynomial weights, let us recall few examples which are recurrent patterns of CTMCs:

- in queuing networks, the policy of a server may be the infinite server policy leading to linear weights;
- in biological and epidemiological models, the rate of some "synchronization" between two instances of some species is quadratic w.r.t. the size of the species.

Probabilistic models. Generally given some probabilistic model and some kind of target set of states, it may occur that some instances of the model are decisive and some others are not. This raises the issue of the decidability status of the decisiveness problem.

The first definition of pPDA seems to be given by Eugene S. Santos [24] in 1972. The CRP and more generally, the qualitative and quantitative model checking has been shown decidable for pPDA (see surveys of Kucera et al. [16] and Brazdil et al. [12]). As we did in the paper, Brazdil et al. [12] and Lin [21] studied the same subclasses of pPDA (called there stateless pPDA and POC). Interestingly, the decidability of the decisiveness property has only be studied and shown decidable for pPDA with constant weights [16]: static pPDA, and even static one-counter automata, are not decisive w.r.t. regular languages but decisiveness is decidable (w.r.t regular languages).

Probabilistic lossy channel systems (pLCS) have been introduced by Iyer et al. [19] in 1997 where they prove that the CRP and the quantitative model checking against a fragment of LTL is decidable. See also Baier et al. [6], Abdulla et al. [1], Rabinovich [23], Bertrand et al. [10], Abdulla et al. [2] for pLCS with modified semantics of losses. Observe that depending on the selected semantics, the model of pLCS leads either to static or dynamic weights. In particular the decisiveness property of pLCS is ensured by a particular case of dynamic weights.

Probabilistic counter machines (pCM) have been studied in [13]. For example, static Probabilistic Petri nets (pPN) are decisive w.r.t. upward-closed sets but it is unknown whether decisiveness is decidable w.r.t. finite sets or w.r.t. dynamic weights.

Our contributions.

- In order to unify analysis of decisiveness, we introduce a dynamic probabilistic version of counter machine (pCM) and we first establish that decisiveness is undecidable for pCMs even with constant weights.
- Then we study the decidability of decisiveness of one-counter pCMs. We show that, without restrictions on dynamic weights, decisiveness is undecidable for one-counter pCM even with a single state. On the contrary, with polynomial weights, decisiveness becomes decidable for a large subclass of one-counter pCMs, called homogeneous probabilistic counter machine (pHM).
- Then we show that decisiveness of probabilistic Petri nets (pPNs) with polynomial weights is undecidable when the target set is finite or upward-closed (unlike the case of constant weights). Finally we prove that the standard subclass of pPNs with a regular language is decisive with respect to a finite set whatever the kind of weights.
- Some of our results are not only technically involved but contain new ideas. In particular, the proof of undecidability of decisiveness for pPN with polynomial weights with respect to a finite or upward closed set is based on an original weak simulation of CM. Similarly the model of pHM can be viewed as a dynamic extension of quasi-birth–death processes well-known in the performance evaluation field [8].

Organisation. Section 2 recalls decisive Markov chains, presents the classical algorithm for solving the CRP problem and shows that decisiveness is somehow related to recurrence of Markov chains. In section 3, we introduce pCM and show that decisiveness is undecidable for static pCM. In section 4, we study the decidability status of decisiveness for probabilistic one-counter pCM and in section 5, the decidability status of decisiveness for pPN. Finally in Section 6 we conclude and give some perspectives to this work. All missing proofs can be found in [17].

2 Decisive Markov chains

As usual, $\mathbb{N}$ and $\mathbb{N}^*$ denote respectively the set of non negative integers and the set of positive integers. The notations $\mathbb{Q}$, $\mathbb{Q}_{\geq 0}$ and $\mathbb{Q}_{>0}$ denote the set of rationals, non-negative rationals and positive rationals. Let $F \subseteq E$; when there is no ambiguity about E, $\overline{F}$ will denote $E \setminus F$.

2.1 Markov chains: definitions and properties

Notations. A set S is *countable* if there exists an injective function from S to the set of natural numbers: hence it could be finite or countably infinite. Let S be a countable set of elements called states. Then $\mathit{Dist}(S) = \{\Delta : S \to \mathbb{Q}_{\geq 0} \mid \sum_{s \in S} \Delta(s) = 1\}$ is the set of *rational distributions* over S. Let $\Delta \in \mathit{Dist}(S)$, then $\mathit{Supp}(\Delta) = \Delta^{-1}(\mathbb{Q}_{>0})$.

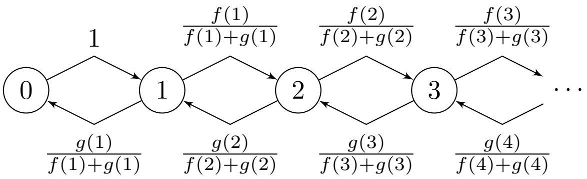


Figure 1 A Markov chain $\mathcal{M}_1$ with for all $n \in \mathbb{N}$, $0 < f(n)$ and $0 < g(n)$.

▶ **Definition 1** ((Effective) Markov chain). *A Markov chain $\mathcal{M} = (S, p)$ is a tuple where:*

- *S is a countable set of states;*
- *p is the transition function from S to $Dist(S)$.*

When for all $s \in S$, $Supp(p(s))$ is finite with both $Supp(p(s))$ and the function $s \mapsto p(s)$ being computable, one says that $\mathcal{M}$ is effective.

When S is countably infinite, we say that $\mathcal{M}$ is *infinite* and we sometimes identify S with $\mathbb{N}$. We also denote $p(s)(s')$ by $p(s, s')$ and $p(s, s') > 0$ by $s \xrightarrow{p(s,s')} s'$. A Markov chain is also viewed as a transition system whose transition relation $\rightarrow$ is defined by $s \rightarrow s'$ if $p(s, s') > 0$.

▶ **Example 2.** Let $\mathcal{M}_1$ be the Markov chain of Figure 1. In any state $i > 0$, the probability for going to the "right", $p(i, i+1) = \frac{f(i)}{f(i)+g(i)}$ and for going to the "left", $p(i, i-1) = \frac{g(i)}{f(i)+g(i)}$. In state 0, one goes to 1 with probability 1. $\mathcal{M}_1$ is effective if the functions f and g are computable.

We denote $\rightarrow^*$, the reflexive and transitive closure of $\rightarrow$ and we say that s' is *reachable from* s if $s \rightarrow^* s'$. We say that a subset $A \subseteq S$ is *reachable* from s if some $s' \in A$ is reachable from s and we denote $s \rightarrow^* A$. Let us remark that every finite path of $\mathcal{M}$ can be extended into (at least) one infinite path.

Given an initial state s_0, the *sampling* of a Markov chain $\mathcal{M}$ is an *infinite random sequence of states* (i.e., a path) $\sigma = s_0 s_1 \ldots$ such that for all $i \geq 0$, $s_i \rightarrow s_{i+1}$. As usual, the corresponding σ-algebra is generated by the finite prefixes of infinite paths and the probability of a measurable subset Π of infinite paths, given an initial state s_0, is denoted $\mathbf{Pr}_{\mathcal{M},s_0}(\Pi)$. In particular denoting $s_0 \ldots s_n S^\omega$ the set of infinite paths with $s_0 \ldots s_n$ as prefix $\mathbf{Pr}_{\mathcal{M},s_0}(s_0 \ldots s_n S^\omega) = \prod_{0 \leq i < n} p(s_i, s_{i+1})$.

Notations. From now on, $\mathbf{G}$ (resp. $\mathbf{F}$, $\mathbf{X}$) denotes the always (resp. eventual, next) operator of LTL, and $\mathbf{E}$ the existential operator of CTL* [7].

Let $A \subseteq S$. We say that σ *reaches* A if $\exists i \in \mathbb{N}$ $s_i \in A$ which corresponds to $\sigma \models \mathbf{F}A$. Similarly $\sigma \models \mathbf{XF}A$ if $\exists i > 0$ $s_i \in A$. The probability that starting from s_0, the path σ reaches A is thus denoted by $\mathbf{Pr}_{\mathcal{M},s_0}(\mathbf{F}A)$.

The next definition states qualitative and quantitative properties of a Markov chain.

▶ **Definition 3** (Irreducibility, recurrence, transience). *Let $\mathcal{M} = (S, p)$ be a Markov chain and $s \in S$. Then:*

- *$\mathcal{M}$ is* irreducible *if for all $s, s' \in S$, $s \rightarrow^* s'$;*
- *s is* recurrent *if $\mathbf{Pr}_{\mathcal{M},s}(\mathbf{XF}\{s\}) = 1$ otherwise s is* transient.

The next proposition states that in an irreducible Markov chain, all states are in the same category [20].

▶ **Proposition 4.** *Let $\mathcal{M} = (S, p)$ be an irreducible Markov chain and $s, s' \in S$. Then s is recurrent if and only if s' is recurrent.*

Thus an irreducible Markov chain will be said transient or recurrent depending on the category of its states (all states are in the same category). In the remainder of this section, we will relate this category with techniques for computing reachability probabilities.

▶ **Example 5.** $\mathcal{M}_1$ of Figure 1 is clearly irreducible. Let us define $p_n = \frac{f(n)}{f(n)+g(n)}$. Then (see [17] for more details), $\mathcal{M}_1$ is recurrent if and only if $\sum_{n\in\mathbb{N}} \prod_{1\leq m<n} \rho_m = \infty$ with $\rho_m = \frac{1-p_m}{p_m}$, and when transient, the probability that starting from i the random path reaches 0 is equal to $\frac{\sum_{i\leq n} \prod_{1\leq m<n} \rho_m}{\sum_{n\in\mathbb{N}} \prod_{1\leq m<n} \rho_m}$.

2.2 Decisive Markov chains

One of the goals of the quantitative analysis of infinite Markov chains is to approximately compute reachability probabilities. Let us formalize it. Given a finite representation of a subset $A \subseteq S$, one says that this representation is *effective* if one can decide the membership problem for A. With a slight abuse of language, we identify A with any effective representation of A.

The Computing of Reachability Probability (CRP) problem

- Input: an effective Markov chain $\mathcal{M}$, an (initial) state s_0, an effective subset of states A, and a rational $\theta > 0$.
- Output: an interval $[low, up]$ such that $up - low \leq \theta$ and $\mathbf{Pr}_{\mathcal{M},s_0}(\mathbf{F}A) \in [low, up]$.

In finite Markov chains, there is a well-known algorithm for computing exactly the reachability probabilities in polynomial time [7]. In infinite Markov chains, there are (at least) two possible research directions: (1) either using the specific features of a formalism to design such a CRP algorithm [16], (2) or requiring a supplementary property on Markov chains in order to design an "abstract" algorithm, then verifying that given a formalism this property is satisfied and finally transforming this algorithm into a concrete one. *Decisiveness*-based approach follows the second direction [3]. In words, decisiveness w.r.t. s_0 and A means that almost surely the random path σ starting from s_0 will reach A or some state s' from which A is unreachable.

▶ **Definition 6.** *A Markov chain $\mathcal{M}$ is* decisive *w.r.t. $s_0 \in S$ and $A \subseteq S$ if:*

$$\mathbf{Pr}_{\mathcal{M},s_0}(\mathbf{G}(\overline{A} \cap \mathbf{EF}A)) = 0$$

Then under the hypotheses of decisiveness w.r.t. s_0 and A and decidability of the reachability problem w.r.t. A, Algorithm 1 solves the CRP problem.

Let us explain Algorithm 1. If A is unreachable from s_0, then it returns the singleton interval $[0, 0]$. Otherwise it maintains a lower bound *pmin* (initially 0) and an upper bound *pmax* (initially 1) of the reachability probability and builds some prefix of the infinite execution tree of $\mathcal{M}$. It also maintains the probability to reach a vertex in this tree. There are three possible cases when examining the state s associated with the current vertex along a path of probability q: (1) either $s \in A$ and the lower bound is incremented by q, (2) either A is unreachable from s and the upper bound is decremented by q, (3) or it extends the prefix of the tree by the successors of s. The lower bound always converges to the searched probability while due to the decisiveness property, the upper bound also converges to it ensuring termination of the algorithm. For the sake of termination, a fair extraction policy is required such as a FIFO one.

▶ **Proposition 7** ([3]). *Algorithm 1 terminates and computes an interval of length at most θ containing $\mathbf{Pr}_{\mathcal{M},s_0}(\mathbf{F}A)$ when applied to a decisive Markov chain $\mathcal{M}$ w.r.t. s_0 and A with a decidable reachability problem w.r.t. A.*

Algorithm 1 can be applied to probabilistic Lossy Channel Systems (pLCS) since they are decisive (Corollary 4.7 in [3] and see [5] for the first statement) and reachability is decidable in LCS [4]. It can be also applied to pVASSs w.r.t. upward closed sets because Corollary 4.4 in [3] states that pVASSs are decisive w.r.t. *upward closed sets.*

Observations. The test $pmin = 0$ is not necessary but adding it avoids to return 0 as lower bound, which would be inaccurate since entering this loop means that A is reachable from s_0. Extractions from the front are performed in a way that the execution tree will be covered (for instance by a breadth first exploration).

Algorithm 1 Framing the reachability probability in decisive Markov chains.

```
CompProb($\mathcal{M}, s_0, A, \theta$)
if not $s_0 \to^* A$ then  return $(0,0)$
$pmin \leftarrow 0$; $pmax \leftarrow 1$; Front $\leftarrow \emptyset$
Insert(Front, $(s_0, 1)$)
while $pmax - pmin > \theta$ or $pmin = 0$ do
    $(s, q) \leftarrow$ Extract(Front)
    if $s \in A$ then $pmin \leftarrow pmin + q$
    else if not $s \to^* A$ then $pmax \leftarrow pmax - q$
    else
        for $s' \in Supp(p(s))$ do
            Insert(Front, $(s', qp(s,s'))$)
        end
    end
end
return $(pmin, pmax)$
```

Let $\mathcal{M}$ be a Markov chain. One denotes $Post^*_{\mathcal{M}}(A)$, the set of states that can be reached from some state of A and $Pre^*_{\mathcal{M}}(A)$, the set of states that can reach A. While decisiveness has been used in several contexts including uncountable probabilistic systems [9], its relation with standard properties of Markov chains has not been investigated. This is the goal of the next definition and proposition. In words, $\mathcal{M}_{s_0,A}$ consists of reachable states not in A but that can reach A with an additional state $s_\perp$ corresponding to states of $\mathcal{M}$ that are either in A or cannot reach A. The probabilities are defined similarly as those of $\mathcal{M}$ except that the transition probabilities to $s_\perp$ are the sums of the transition probabilities to the corresponding states in $\mathcal{M}$.

▶ **Definition 8.** *Let $\mathcal{M}$ be a Markov chain, $s_0 \in S$ and $A \subseteq S$ such that $s_0 \in Pre^*_{\mathcal{M}}(A) \setminus A$. The Markov chain $\mathcal{M}_{s_0,A} = (S_{s_0,A}, p_{s_0,A})$ is defined as follows:*

- *$S_{s_0,A}$ is the union of (1) the smallest set containing s_0 and such that for all $s \in S_{s_0,A}$ and $s' \in Pre^*_{\mathcal{M}}(A) \setminus A$ with $s \to s'$, one have:*
  $s' \in S_{s_0,A}$ and (2) $\{s_\perp\}$ where $s_\perp$ is a new state;
- *for all $s, s' \neq s_\perp$, $p_{s_0,A}(s,s') = p(s,s')$ and $p_{s_0,A}(s,s_\perp) = \sum_{s' \notin \in Pre^*_{\mathcal{M}}(A) \setminus A} p(s,s')$;*
- *$p_{s_0,A}(s_\perp, s_0) = 1$.*

▶ **Proposition 9.** *Let $\mathcal{M} = (S, p)$ be a Markov chain, $s_0 \in S$ and $A \subseteq S$ such that $s_0 \in Pre^*_{\mathcal{M}}(A) \setminus A$. Then $\mathcal{M}_{s_0,A}$ is irreducible. Furthermore $\mathcal{M}$ is decisive w.r.t. s_0 and A if and only if $\mathcal{M}_{s_0,A}$ is recurrent.*

Proof. Let $s \in S_{s_0,A} \setminus \{s_\perp\}$. Then s is reachable from s_0 and A is reachable from s in $\mathcal{M}$ implying that $s \rightarrow^* s_\perp$ in $\mathcal{M}_{s_0,A}$ (using a shortest path for reachability). Since $s_\perp \rightarrow s_0$, $s_\perp \rightarrow^* s$. Thus $\mathcal{M}_{s_0,A}$ is irreducible.
$\mathcal{M}_{s_0,A}$ is recurrent iff $\mathbf{Pr}_{\mathcal{M}_{s_0,A},s_\perp}(\mathbf{XF}\{s_\perp\}) = 1$ iff $\mathbf{Pr}_{\mathcal{M}_{s_0,A},s_0}(\mathbf{F}\{s_\perp\}) = 1$ iff $\mathbf{Pr}_{\mathcal{M},s_0}(\mathbf{F}A \cup \overline{Pre^*_{\mathcal{M}}(A)}) = 1$ iff $\mathcal{M}$ is decisive w.r.t. s_0 and A. ◀

The equivalence between decisiveness of $\mathcal{M}$ w.r.t. $s_0 \in S$ and $A \subseteq S$ and recurrence of $\mathcal{M}_{s_0,A}$ allows to apply standard criteria for recurrence in order to check decisiveness. For instance in Section 4, we will use the criterion presented in Example 5 for the Markov chain of Figure 1.

3 Probabilistic counter machines

We now introduce *probabilistic Counter Machines (pCM)* in order to study the decidability of the decisiveness property w.r.t. several relevant subclasses of pCM.

▶ **Definition 10** (pCM). *A* probabilistic counter machine (pCM) *is a tuple $\mathcal{C} = (Q, P, \Delta, W)$ where:*

- *Q is a finite set of control states;*
- *$P = \{p_1, \ldots, p_d\}$ is a finite set of counters (also called places);*
- *$\Delta = \Delta_0 \uplus \Delta_1$ where Δ_0 is a finite subset of $Q \times P \times \mathbb{N}^d \times Q$ and Δ_1 is a finite subset of $Q \times \mathbb{N}^d \times \mathbb{N}^d \times Q$;*
- *W is a computable function from $\Delta \times \mathbb{N}^d$ to $\mathbb{N}^*$.*

Notations. A transition $t \in \Delta_0$ is denoted $t = (q_t^-, p_t, \mathbf{Post}(t), q_t^+)$ and also $q_t^- \xrightarrow{p_t, \mathbf{Post}(t)} q_t^+$. A transition $t \in \Delta_1$ is denoted $t = (q_t^-, \mathbf{Pre}(t), \mathbf{Post}(t), q_t^+)$ and also $q_t^- \xrightarrow{\mathbf{Pre}(t), \mathbf{Post}(t)} q_t^+$. Let t be a transition of $\mathcal{C}$. Then $W(t)$ is the function from $\mathbb{N}^d$ to $\mathbb{Q}_{>0}$ defined by $W(t)(\mathbf{m}) = W(t, \mathbf{m})$. A polynomial is *positive* if all its coefficients are non-negative and there is a positive constant term. When for all $t \in T$, $W(t)$ is a positive polynomial whose variables are the counters, we say that $\mathcal{C}$ is a *polynomial* pCM.

A *configuration* of $\mathcal{C}$ is an item of $Q \times \mathbb{N}^d$. Let $s = (q, \mathbf{m})$ be a configuration and $t = (q_t^-, p_t, \mathbf{Post}(t), q_t^+)$ be a transition in Δ_0. Then t is *enabled* in s if $\mathbf{m}(p_t) = 0$ and $q = q_t^-$; its *firing* leads to the configuration $(q_t^+, \mathbf{m} + \mathbf{Post}(t))$. Let $t = (q_t^-, \mathbf{Pre}(t), \mathbf{Post}(t), q_t^+) \in \Delta_1$. Then t is *enabled* in s if $\mathbf{m} \geq \mathbf{Pre}(t)$ and $q = q_t^-$; its *firing* leads to the configuration $s' = (q_t^+, \mathbf{m} - \mathbf{Pre}(t) + \mathbf{Post}(t))$. One denotes the configuration change by: $s \xrightarrow{t} s'$. One denotes $En(s)$, the set of transitions enabled in s and $Weight(s) = \sum_{t \in En(s)} W(t, \mathbf{m})$. Let $\sigma = t_1 \ldots t_n$ be a sequence of transitions. We define the enabling and the firing of σ by induction. The empty sequence is always enabled in s and its firing leads to s. When $n > 0$, σ is enabled if $s \xrightarrow{t_1} s_1$ and $t_2 \ldots t_n$ is enabled in s_1. The firing of σ leads to the configuration reached by $t_2 \ldots t_n$ from s_1. A configuration s is *reachable* from some s_0 if there is a firing sequence σ that reaches s from s_0. When Q is a singleton, one omits the control states in the definition of transitions and configurations.

We now provide the semantic of a pCM as a countable Markov chain.

▶ **Definition 11.** *Let $\mathcal{C}$ be a pCM. Then the Markov chain $\mathcal{M}_{\mathcal{C}} = (S, p)$ is defined by:*

- $S = Q \times \mathbb{N}^d$*;*
- *For all $s = (q, \mathbf{m}) \in S$, if $En(s) = \emptyset$ then $p(s, s) = 1$. Otherwise for all $s' \in S$:*

$$p(s, s') = Weight(s)^{-1} \sum_{s \xrightarrow{t} s'} W(t, \mathbf{m})$$

For establishing the undecidability results, we will reduce an undecidable problem related to *counter programs*, which are a variant of CM. Let us recall that a *d-counter program* $\mathcal{P}$ is defined by a set of d counters $\{c_1, \ldots, c_d\}$ and a set of $n + 1$ instructions labelled by $\{0, \ldots, n\}$, where for all $i < n$, the instruction i is of type

- either (1) $c_j \leftarrow c_j + 1;$ **goto** i' with $1 \leq j \leq d$ and $0 \leq i' \leq n$
- or (2) **if** $c_j > 0$ **then** $c_j \leftarrow c_j - 1;$ **goto** i', **else goto** i'' with $1 \leq j \leq d$ and $0 \leq i', i'' \leq n$

and the instruction n is **halt**. The program starts at instruction 0 and halts if it reaches the instruction n.

The halting problem for two-counter programs asks, given a two-counter program $\mathcal{P}$ and initial values of counters, whether $\mathcal{P}$ eventually halts. It is undecidable [22]. We introduce a subclass of two-counter programs that we call *normalized.* A normalized two-counter program $\mathcal{P}$ starts by resetting its counters and, on termination, resets its counters before halting.

Normalized two-counter program. The first two instructions of a normalized two-counter program reset counters c_1, c_2 as follows:

- $0:$ **if** $c_1 > 0$ **then** $c_1 \leftarrow c_1 - 1;$ **goto** 0 **else goto** 1
- $1:$ **if** $c_2 > 0$ **then** $c_2 \leftarrow c_2 - 1;$ **goto** 1 **else goto** 2

The last three instructions of a normalized two-counter program are:

- $n-2:$ **if** $c_1 > 0$ **then** $c_1 \leftarrow c_1 - 1;$ **goto** $n-2$ **else goto** $n-1$
- $n-1:$ **if** $c_2 > 0$ **then** $c_2 \leftarrow c_2 - 1;$ **goto** $n-1$ **else goto** n
- $n:$ **halt**

For $1 < i < n - 2$, the labels occurring in instruction i belong to $\{0, \ldots, n - 2\}$. In a normalized two-counter program $\mathcal{P}$, given any initial values v_1, v_2, $\mathcal{P}$ halts with v_1, v_2 if and only if $\mathcal{P}$ halts with initial values $0, 0$. Moreover when $\mathcal{P}$ halts, the values of the counters are null. The halting problem for normalized two-counter programs is also undecidable (see [17] for the proof).

We now show that decisiveness is undecidable even for *static* pCM, by considering only *static* weights: for all $t \in \Delta$, $W(t)$ is a constant function.

▶ **Theorem 12.** *Decisiveness w.r.t. a finite set is undecidable in (static) pCM.*

4 Probabilistic safe one-counter machines

We now study decisiveness for pCMs that only have one counter denoted c. We also restrict Δ_1: a single counter PCM is *safe* if for all $t \in \Delta_1$, $(\mathbf{Pre}(t), \mathbf{Post}(t)) \in \{1\} \times \{0, 1, 2\}$. In words, in a safe one-counter pCM, a transition of Δ_1 requires the counter to be positive and may either let it unchanged, or incremented or decremented by a unit.

4.1 One-state and one-counter pCM

We first prove that decisiveness is undecidable for the probabilistic version of one-state and one-counter machines. Then we show how to restrict the weight functions and Δ_1 such that this property becomes decidable. Both proofs make use of the relationship between decisiveness and recurrence stated in Proposition 9, in an implicit way.

▶ **Theorem 13.** *The decisiveness problem for safe one-counter pCM is undecidable even with a single state.*

Proof. We will reduce the Hilbert's tenth problem to decisiveness problems. Let $P \in \mathbb{Z}[X_1, \dots X_k]$ be an integer polynomial with k variables. This problem asks whether there exist $n_1, \dots, n_k \in \mathbb{N}$ such that $P(n_1, \dots, n_k) = 0$.

We define $\mathcal{C}$ as follows. There are two transitions both in Δ_1:

- dec with $\mathbf{Pre}(dec) = 1$ and $\mathbf{Post}(dec) = 0$;
- inc with $\mathbf{Pre}(inc) = 0$ and $\mathbf{Post}(inc) = 1$.

The weight of dec is the constant function 1, i.e., $W(dec, n) = f(n) = 1$, while the weight of inc is defined by the following (non polynomial) function:

$$W(inc, n) = g(n) = \min(P^2(n_1, \dots, n_k) + 1 \mid n_1 + \dots + n_k \leq n)$$

This function is obviously computable. Let us study the decisiveness of $\mathcal{M}_\mathcal{C}$ w.r.t. $s_0 = 1$ and $A = \{0\}$. Observe that $\mathcal{M}_\mathcal{C}$ is the Markov chain $\mathcal{M}_1$ of Figure 1. Let us recall that in $\mathcal{M}_1$, the probability to reach 0 from i is 1 iff $\sum_{n \in \mathbb{N}} \prod_{1 \leq m < n} \rho_m = \infty$ and otherwise it is equal to $\frac{\sum_{i \leq n} \prod_{1 \leq m < n} \rho_m}{\sum_{n \in \mathbb{N}} \prod_{1 \leq m < n} \rho_m}$ with $\rho_m = \frac{1 - p_m}{p_m}$.

• Assume there exist $n_1, \dots, n_k \in \mathbb{N}$ s.t. $P(n_1, \dots, n_k) = 0$. Let $n_0 = n_1 + \dots + n_k$. Thus for all $n \geq n_0$, $W(inc, n) = 1$, which implies that $p_\mathcal{C}(n, n-1) = p_\mathcal{C}(n, n+1) = \frac{1}{2}$. Thus due to the results on $\mathcal{M}_1$, from any state n, one reaches 0 almost surely and so $\mathcal{M}_\mathcal{C}$ is decisive.

• Assume there do not exist $n_1, \dots, n_k \in \mathbb{N}$ s.t. $P(n_1, \dots, n_k) = 0$. For all $n \in \mathbb{N}$, $W(inc, n) \geq 2$, implying that in $\mathcal{M}_1$, $\rho_n \leq \frac{1}{2}$. Thus $\mathcal{M}_\mathcal{C}$ is not decisive. ◀

Due to the negative result for single state and single counter pCM stated in Theorem 13, it is clear that one must restrict the possible weight functions.

▶ **Theorem 14.** *The decisiveness problem w.r.t. s_0 and finite A for polynomial safe one-counter pCM $\mathcal{C}$ with a single state is decidable in linear time.*

4.2 Homogeneous one-counter machines

We now study another interesting model that is considered as a generalization of the well-known model of quasi-birth-death process with dynamic weights. This model has been the topic of numerous theoretical results and modelings, see for instance [8].

Let $\mathcal{C}$ be a one-counter safe pCM. For all $q \in Q$, let $S_{q,1} = \sum_{t=(q,\mathbf{Pre}(t),\mathbf{Post}(t),q_t^+) \in \Delta_1} W(t)$ and $\mathbf{M}_\mathcal{C}$ be the $Q \times Q$ matrix defined by

$$\mathbf{M}_\mathcal{C}[q, q'] = \frac{\sum_{t=(q,\mathbf{Pre}(t),\mathbf{Post}(t),q') \in \Delta_1} W(t)}{S_{q,1}}$$

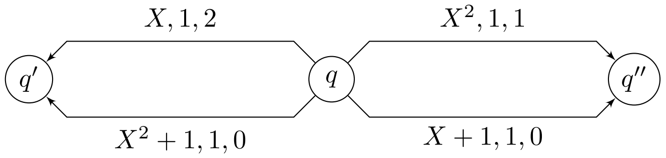
(thus $\mathbf{M}_\mathcal{C}[q, q']$ is a function from $\mathbb{N}$ to $\mathbb{Q}_{\geq}0$).

▶ **Definition 15** (pHM). *A* probabilistic homogeneous machine *(pHM) is a probabilistic safe one-counter machine $\mathcal{C} = (Q, \Delta, W)$ where:*

- *For all $t \in \Delta$, $W(t)$ is a positive polynomial in $\mathbb{N}[X]$;*
- *For all $q, q' \in Q$, $\mathbf{M}_\mathcal{C}[q, q']$ is constant.*

Observe that by definition, in a pHM, $\mathbf{M}_\mathcal{C}$ is a transition matrix.

▶ **Example 16.** Here $\mathbf{M}_\mathcal{C}[q, q'] = \mathbf{M}_\mathcal{C}[q, q''] = \frac{X^2+X+1}{2(X^2+X+1)} = \frac{1}{2}$ fulfilling the homogeneous requirement. Let us describe a possible transition: given a configuration (q, n) with $n > 0$ the probability to go to $(q', n+1)$ is equal to $\frac{X}{2(X^2+X+1)}$.



The family $(r_q)_{q \in Q}$ of the next proposition is independent of the function W and is associated with the qualitative behaviour of $\mathcal{C}$, i.e., its underlying transition system. For all $q \in Q$, r_q is an upper bound of the counter value for q, from which $Q \times \{0\}$ is reachable.

▶ **Proposition 17.** *Let $\mathcal{C}$ be a pHM. Then one can compute in polynomial time a family $(r_q)_{q \in Q}$ such that for all q, $r_q \in \{0, \ldots, |Q| - 1\} \cup \{\infty\}$, and $Q \times \{0\}$ is reachable from (q, k) iff $k \leq r_q$.*

▶ **Theorem 18.** *Let $\mathcal{C}$ be a pHM such that $\mathbf{M}_\mathcal{C}$ is irreducible. Then the decisiveness problem of $\mathcal{C}$ w.r.t. $s_0 = (q, n) \in Q \times \mathbb{N}$ and $A = Q \times \{0\}$ is decidable in polynomial time.*

Proof. With the notations of previous proposition, assume that there exist q with $r_q < \infty$ and q' with $r_{q'} = \infty$. Since $\mathbf{M}_\mathcal{C}$ is irreducible, there is a sequence of transitions in Δ_1 $q_0 \xrightarrow{1, v_1} q_1 \cdots \xrightarrow{1, v_m} q_m$ with $q_0 = q$ and $q_m = q'$. Let $sv = \min(\sum_{i \leq j} (v_i - 1) | j \leq m)$ and pick some $k > \max(r_q, -sv)$. Then there is a path in $\mathcal{M}_\mathcal{C}$ from (q, k) to $(q', k + \sum_{i \leq m} (v_i - 1))$, which yields a contradiction since (q, k) cannot reach $Q \times \{0\}$ while $(q', k + \sum_{i \leq m} v_i)$ can reach it. Thus either (1) for all $q \in Q$, $r_q < \infty$ or (2) for all $q \in Q$, $r_q = \infty$.

• First assume that for all $q \in Q$, $r_q < \infty$. Thus for all $k > r_q$, (q, k) cannot reach $Q \times \{0\}$ and thus $\mathcal{C}$ is decisive w.r.t. (q, k) and $Q \times \{0\}$. Now consider a configuration (q, k) with $k \leq r_q$. By definition there is a positive probability say $p_{(q,k)}$ to reach $Q \times \{0\}$ from (q, k). Let $p_{\min} = \min(p_{(q,k)} \mid q \in Q \wedge k \leq r_q)$. Then for all (q, k) with $k \leq r_q$, there is a probability at least $p_{\min}$ to reach either $Q \times \{0\}$ or $\{(q, k) \mid q \in Q \wedge k > r_q\}$ by a path of length $\ell = \sum_{q \in Q} (r_q + 1)$. This implies that after $n\ell$ transitions the probability to reach either $Q \times \{0\}$ or $\{(q, k) \mid q \in Q \wedge k > r_q\}$ is at least $1 - (1 - p_{\min})^n$. Thus $\mathcal{C}$ is decisive w.r.t. (q, k) and $Q \times \{0\}$. Summarizing for all (q, k), $\mathcal{C}$ is decisive w.r.t. (q, k) and $Q \times \{0\}$.

• Now assume that for all $(q, k) \in Q \times \mathbb{N}$, $Q \times \{0\}$ is reachable from (q, k). Thus the decisiveness problem boils down to the almost sure reachability of $Q \times \{0\}$.

Since the target of decisiveness is $Q \times \{0\}$, we can arbitrarily set up the outgoing transitions of these states (i.e., Δ_0) without changing the decisiveness problem. So we choose these transitions and associated probabilities as follows. For all q, q' such that $\mathbf{M}_\mathcal{C}[q, q'] > 0$, there is a transition $t = q \xrightarrow{c, 0} q'$ with $W(t) = \mathbf{M}_\mathcal{C}[q, q']$.

Since $\mathbf{M}_\mathcal{C}$ is irreducible, there is a unique invariant distribution π_∞ (i.e., $\pi_\infty \mathbf{M}_\mathcal{C} = \pi_\infty$) fulfilling for all $q \in Q$, $\pi_\infty(q) > 0$.

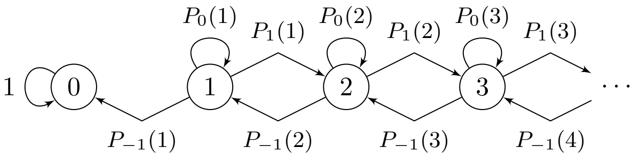
Let $(Q_n, N_n)_{n \in \mathbb{N}}$ be the stochastic process defined by $\mathcal{M}_\mathcal{C}$ with $N_0 = k$ for some k and for all $q \in Q$, $\mathbf{Pr}(Q_0 = q) = \pi_\infty(q)$. Due to the invariance of π_∞ and the choice of transitions for $Q \times \{0\}$, one gets by induction that for all $n \in \mathbb{N}$:

- $\mathbf{Pr}(Q_n = q) = \pi_\infty(q)$;
- for all $k > 0$ and $v \in \{-1, 0, 1\}$, $\mathbf{Pr}(N_{n+1} = k + v - 1 | N_n = k) =$ $\sum_{q \in Q} \pi_\infty(q) \frac{\sum_{t=(q,1,v,q') \in \Delta_1} W(t,k)}{S_{q,1}(k)} = \frac{\sum_{q \in Q} \pi_\infty(q) \prod_{q' \neq q} S_{q',1}(k) \sum_{t=(q,1,v,q') \in \Delta} W(t,k)}{\prod_{q' \in Q} S_{q',1}(k)}$;
- $\mathbf{Pr}(N_{n+1} = 0 | N_n = 0) = 1$.

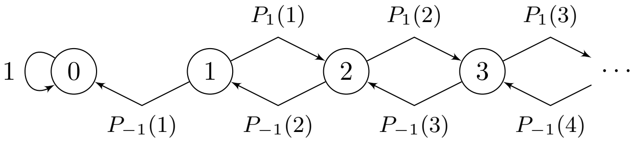
For $v \in \{-1, 0, 1\}$, let us define the polynomial P_v by:

$$\sum_{q \in Q} \pi_\infty(q) \prod_{q' \neq q} S_{q',1} \sum_{t=(q,1,v+1,q') \in \Delta_1} W(t)$$

Due to the previous observations, the stochastic process $(N_n)_{n\in\mathbb{N}}$ is the Markov chain defined below where the weights outgoing from a state have to be normalized:



Using our hypothesis about reachability, P_{-1} is a positive polynomial (while P_1 could be null) and thus the decisiveness of this Markov chain w.r.t. state 0 is equivalent to the decisiveness of the Markov chain below:



Due to Theorem 14, this problem is decidable (in linear time) and either (1) for all $k \in \mathbb{N}$ this Markov chain is decisive w.r.t k and 0 or (2) for all $k > 0$ this Markov chain is not decisive w.r.t k and 0. Let us analyze the two cases w.r.t. the Markov chain of the pHM.

Case (1). In the stochastic process $(Q_n, N_n)_{n\in\mathbb{N}}$, the initial distribution has a positive probability for (q, k) for all $q \in Q$. This implies that for all q, $\mathcal{C}$ is decisive w.r.t. (q, k) and $Q \times \{0\}$. Since k was arbitrary, this means that for all (q, k), $\mathcal{C}$ is decisive w.r.t. (q, k) and $Q \times \{0\}$.

Case (2). Choosing $k = 1$ and applying the same reasoning as for the previous case, there is some $(q, 1)$ which is not decisive (and so for all (q, k') with $k' > 0$). Let $q' \in Q$, since $\mathbf{M}_{\mathcal{C}}$ is irreducible, there is a (shortest) sequence of transitions in Δ_1 leading from q' to q whose length is at most $|Q| - 1$. Thus for all (q', k') with $k' \geq |Q|$ there is a positive probability to reach some (q, k) with $k > 0$. Thus (q', k) is not decisive.
Now let (q', k') with $k' < |Q|$. Then we compute by a breadth first exploration the configurations reachable from (q', k') until either (1) one reaches some (q'', k'') with $k'' \geq |Q|$ or (2) the full (finite) reachability set is computed. In the first case, there is a positive probability to reach some (q'', k'') with $k'' \geq |Q|$ and from (q'', k'') to some (q, k) with $k > 0$ and so (q', k') is not decisive. In the second case, it means that the reachable set is finite and from any configuration of this set there is a positive probability to reach $Q \times \{0\}$ by a path of length at most the size of this set. Thus almost surely $Q \times \{0\}$ will be reached and (q', k') is decisive. ◀

5 Probabilistic Petri nets

We now introduce probabilistic Petri nets as a subclass of pCM.

▶ **Definition 19** (pPN). *A probabilistic Petri net (pPN) $\mathcal{N}$ is a pCM $\mathcal{N} = (Q, P, \Delta, W)$ where Q is a singleton and $\Delta_0 = \emptyset$.*

Notations. Since there is a unique control state in a pPN, a configuration in a pPN is reduced to $\mathbf{m} \in \mathbb{N}^P$ and it is called a *marking*. As usual a marking $\mathbf{m}$ is also denoted as a bag $\sum_{p\in P} \mathbf{m}(p)p$ where the term $\mathbf{m}(p)p$ is omitted when $\mathbf{m}(p) = 0$ and the term $\mathbf{m}(p)p$ is rewritten p when $\mathbf{m}(p) = 1$. A pair $(\mathcal{N}, \mathbf{m}_0)$, where $\mathcal{N}$ is a pPN and $\mathbf{m}_0 \in \mathbb{N}^P$ is some (initial) marking, is called a *marked pPN*. As pPN is defined as a subclass of pCM, its formal semantics is the same as the one described in Section 3.

In previous works [3, 11] about pPNs, the weight function W is a *static* one: i.e., a function from Δ to $\mathbb{N}^*$. As above, we call these models *static* probabilistic Petri nets.

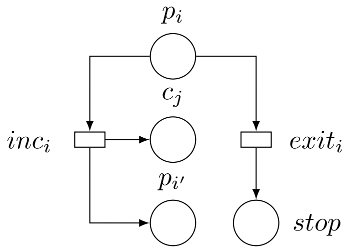


Figure 2 $i : c_j \leftarrow c_j + 1; \textbf{goto } i'$.

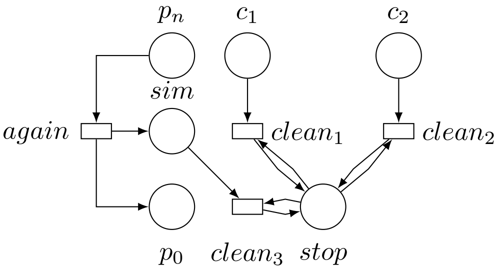


Figure 3 halt instruction and cleaning stage.

Static-probabilistic VASS (and so pPNs) are *decisive* with respect to *upward closed sets* (Corollary 4.4 in [3]) but they may not be decisive w.r.t. an arbitrary finite set. Surprisingly, the decisiveness problem for Petri nets or VASS seems not to have been studied. We establish below that even for polynomial pPNs, decisiveness is undecidable.

▶ **Theorem 20.** *The decisiveness problem of polynomial pPNs w.r.t. a finite or upward closed set is undecidable.*

Proof. We reduce the reachability problem of normalized two-counter machines to the decisiveness problem of pPN. Let $\mathcal{C}$ be a normalized two-counter machine with an instruction set $\{0, \ldots, n\}$. The corresponding marked pPN $(\mathcal{N}_\mathcal{C}, \mathbf{m}_0)$ is built as follows. Its set of places is $P = \{p_i \mid 0 \leq i \leq n\} \cup \{q_i \mid i \text{ is a test instruction}\} \cup \{c_j \mid 1 \leq j \leq 2\} \cup \{sim, stop\}$. The initial marking is $\mathbf{m}_0 = p_0$.

The set Δ of transitions is defined by a pattern per type of instruction. The pattern for the incrementation instruction is depicted in Figure 2. The pattern for the test instruction is depicted in Figure 4. The pattern for the halt instruction is depicted in Figure 3 with in addition a *cleaning stage*. A place is depicted by a circle while a transition is depicted by a rectangle. There is an edge from place p to transition t (resp. from transition t to place p) labelled by $v = \mathbf{Pre}(t)(p)$ (resp. $v = \mathbf{Post}(t)(p)$) when $v > 0$; v is omitted when $v = 1$.

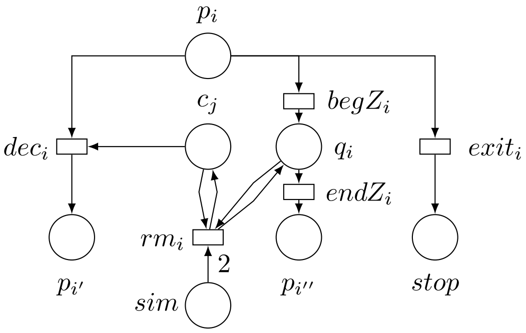


Figure 4 $i : \textbf{if } c_j > 0 \textbf{ then } c_j \leftarrow c_j - 1; \textbf{goto } i' \textbf{else goto } i''$.

Before specifying the weight function W, let us describe the qualitative behaviour of this net. $(\mathcal{N}_\mathcal{C}, \mathbf{m}_0)$ performs repeatedly a *weak* simulation of $\mathcal{C}$. As usual since the zero test does not exist in Petri nets, during a test instruction i, the simulation can follow the zero branch while the corresponding counter is non null (transitions $begZ_i$ and $endZ_i$). If the net has cheated then with transition rm_i, it can remove tokens from sim (two per two). In addition when the instruction is not **halt**, instead of simulating it, it can *exit* the simulation by putting a token in *stop* and then will remove tokens from the counter places including the simulation counter as long as they are not empty. The simulation of the **halt** instruction consists in restarting the simulation and incrementing the simulation counter *sim*.

Thus the set of reachable markings is included in the following set of markings $\{p_i + xc_1 + yc_2 + zsim \mid 0 \leq i \leq n, x, y, z \in \mathbb{N}\} \cup \{q_i + xc_1 + yc_2 + zsim \mid i \text{ is a test instruction}, x, y, z \in \mathbb{N}\} \cup \{stop + xc_1 + yc_2 + zsim \mid x, y, z \in \mathbb{N}\}$. By construction, the marking *stop* is always reachable. We will establish that $\mathcal{N}_\mathcal{C}$ is decisive w.r.t. $\mathbf{m}_0$ and $\{stop\}$ if and only if $\mathcal{C}$ does not halt.

Let us specify the weight function. For any incrementation instruction i, $W(inc_i, \mathbf{m}) = \mathbf{m}(sim)^2 + 1$. For any test instruction i, $W(begZ_i, \mathbf{m}) = \mathbf{m}(sim)^2 + 1$, $W(dec_i, \mathbf{m}) = 2\mathbf{m}(sim)^4 + 2$ and $W(rm_i, \mathbf{m}) = 2$. All other weights are equal to 1.

• Assume that $\mathcal{C}$ halts and consider its execution $\sigma_\mathcal{C}$ with initial values $(0, 0)$. Let $\ell = |\sigma_\mathcal{C}|$ be the length of this execution. Consider now σ the infinite sequence of $(\mathcal{N}_\mathcal{C}, \mathbf{m}_0)$ that infinitely performs the correct simulation of this execution. The infinite sequence σ never marks the place *stop*. We now show that the probability of σ is non null implying that $\mathcal{N}_\mathcal{C}$ is not decisive.

After every simulation of $\sigma_\mathcal{C}$, the marking of *sim* is incremented and it is never decremented since (due to the correctness of the simulation) every time a transition $begZ_i$ is fired, the corresponding counter place c_j is unmarked which forbids the firing of rm_i. So during the $(n+1)^{th}$ simulation of ρ, the marking of *sim* is equal to n.

So consider the probability of the correct simulation of an instruction i during the $(n+1)^{th}$ simulation.

- If i is an incrementation then the weight of inc_i is n^2 and the weight of $exit_i$ is 1. So the probability of a correct simulation is $\frac{n^2+1}{n^2+2} = 1 - \frac{1}{n^2+2} \geq e^{-\frac{2}{n^2+2}}$.[1]
- If i is a test of c_j and the marking of c_j is non null then the weight of dec_i is $2n^4 + 2$, the weight of $begZ_i$ is $n^2 + 1$ and the weight of $exit_i$ is 1. So the probability of a correct simulation is $\frac{2n^4+2}{2n^4+n^2+4} \geq \frac{2n^4+2}{2n^4+2n^2+4} = \frac{n^2+1}{n^2+2} = 1 - \frac{1}{n^2+2} \geq e^{-\frac{2}{n^2+2}}$.
- If i is a test of c_j and the marking of c_j is null then the weight of $begZ_i$ is $n^2 + 1$ and the weight of $exit_i$ is 1. So the probability of a correct simulation is $\frac{n^2+1}{n^2+2} = 1 - \frac{1}{n^2+2} \geq e^{-\frac{2}{n^2+2}}$.

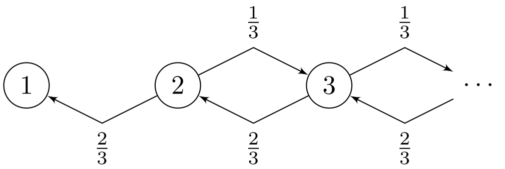
So the probability of the correct simulation during the $(n+1)^{th}$ simulation is at least $(e^{-\frac{2}{n^2+2}})^\ell = e^{-\frac{2\ell}{n^2+2}}$. Hence the probability of σ is at least $\prod_{n \in \mathbb{N}} e^{-\frac{2\ell}{n^2+2}} = e^{-\sum_{n \in \mathbb{N}} \frac{2\ell}{n^2+2}} > 0$, as the sum in the exponent converges.

• Assume that $\mathcal{C}$ does not halt (and so does not halt for any initial values of the counters). We partition the set of infinite paths into a countable family of subsets and prove that for all of them the probability to infinitely avoid to mark *stop* is null which will imply that $\mathcal{N}_\mathcal{C}$ is decisive. The partition is based on $k \in \mathbb{N} \cup \{\infty\}$, the number of firings of *again* in the path.

[1] We use $1 - x \geq e^{-2x}$ for $0 \leq x \leq \frac{1}{2}$.

Case $k < \infty$. Let σ be such a path and consider the suffix of σ after the last firing of *again*. The marking of *sim* is at most k and can only decrease along the suffix. Consider a simulation of an increment instruction i. The weight of inc_i is at most is k^2+1 and the weight of $exit_i$ is 1. So the probability of avoiding $exit_i$ is at most $\frac{k^2+1}{k^2+2} = 1 - \frac{1}{k^2+2} \leq e^{-\frac{1}{k^2+2}}$. Consider the simulation of a test instruction i. Then the weight of dec_i is at most $2k^4+2$, the weight of $begZ_i$ is at most k^2+1 and the weight of $exit_i$ is 1. So the probability of avoiding $exit_i$ is at most $\frac{2k^4+k^2+2}{2k^4+k^2+4} \leq \frac{4k^4+1}{4k^4+2} = 1 - \frac{1}{4k^4+2} \leq e^{-\frac{1}{4k^4+2}}$.
Thus after n simulations of instructions in the suffix, the probability to avoid to mark *stop* is at most $e^{-\frac{n}{4k^4+2}}$. Letting n go to infinity yields the result.

Case $k = \infty$. We first show that almost surely there will be an infinite number of simulations of $\mathcal{C}$ with the marking of *sim* at most 1. Observe that all these simulations are incorrect since they mark p_n while $\mathcal{C}$ does not halt. So at least once per simulation some place q_i and the corresponding counter c_j must be marked and if the marking of *sim* is at least 2 with probability $\frac{2}{3}$ two tokens of *sim* are removed (recall that the weight of rm_i is 2 and the weight of $endZ_i$ is 1). Thus once the marking of *sim* is greater than 1, considering the successive random markings of *sim* after the firing of *again* until it possibly reaches 1, this behaviour is *stochastically bounded* by the following random walk:



In this random walk, one reaches the state 1 with probability 1. This establishes that almost surely there will be an infinite number of simulations of $\mathcal{C}$ with the marking of *sim* at most 1. Such a simulation must simulate at least one instruction. If this instruction is an incrementation, the exiting probability is at least $\frac{1}{3}$; if it is a test instruction the exiting probability is at least $\frac{1}{7}$. Thus after n such simulations of $\mathcal{C}$, the probability to avoid to mark *stop* is at most $(\frac{6}{7})^n$. Letting n go to infinity yields the result.
Observe that the result remains true when substituting the singleton $\{stop\}$ by the set of markings greater than or equal to *stop*. ◀

We deduce thus that decisiveness of extended (probabilistic) Petri nets is undecidable : in particular for Reset Petri nets [15], Post-Self-Modifying Petri nets [25], Recursive Petri nets, etc.

▶ **Definition 21.** *The* language *of a marked Petri net* $(\mathcal{N}, \mathbf{m}_0)$ *is defined by* $\mathcal{L}(\mathcal{N}, \mathbf{m}_0) = \{\sigma \in \Delta^* \mid \mathbf{m}_0 \xrightarrow{\sigma}\}$. *The marked Petri net* $(\mathcal{N}, \mathbf{m}_0)$ *is* regular *if* $\mathcal{L}(\mathcal{N}, \mathbf{m}_0)$ *is regular.*

Given a marked Petri net $(\mathcal{N}, \mathbf{m}_0)$, the problem who asks whether it is regular is decidable [18, 26] and belongs to EXPSPACE [14]. For establishing the next proposition, we only need the following result that holds for regular Petri nets: There exists a computable bound $B(\mathcal{N}, \mathbf{m}_0)$ such that for all markings $\mathbf{m}_1$ reachable from $\mathbf{m}_0$ and all markings $\mathbf{m}_2$ with some $p \in P$ fulfilling $\mathbf{m}_2(p) + B(\mathcal{N}, \mathbf{m}_0) < \mathbf{m}_1(p)$, $\mathbf{m}_2$ is unreachable from $\mathbf{m}_1$ ([18]).

▶ **Theorem 22.** *Let* $(\mathcal{N}, \mathbf{m}_0)$ *be a regular marked pPN and* $\mathbf{m}_1$ *be a marking. Then* $(\mathcal{N}, \mathbf{m}_0)$ *is decisive with respect to* $\mathbf{m}_0$ *and* $\{\mathbf{m}_1\}$.

Proof. Consider the following algorithm that, after computing $B(\mathcal{N}, \mathbf{m}_0)$, builds a finite graph whose vertices are some reachable markings and edges correspond to transition firings between markings:

- The initial vertex is $\mathbf{m}_0$ and push on the stack $\mathbf{m}_0$.
- While the stack is not empty, pop from the stack some marking $\mathbf{m}$. Compute the set of transition firings $\mathbf{m} \xrightarrow{t} \mathbf{m}'$. Add $\mathbf{m}'$ in the set of vertices, the firings $\mathbf{m} \xrightarrow{t} \mathbf{m}'$ to the set of edges and push on the stack $\mathbf{m}'$ if:
  1. $\mathbf{m}'$ is not already present in the set of vertices,
  2. and $\mathbf{m}' \neq \mathbf{m}_1$,
  3. and for all $p \in P$, $\mathbf{m}_1(p) + B(\mathcal{N}, \mathbf{m}_0) \geq \mathbf{m}'(p)$.

Due to the third condition, this algorithm terminates. From above, if $\mathbf{m}_1$ does not occur in the graph then $\mathbf{m}_1$ is unreachable from $\mathbf{m}_0$ and thus $\mathcal{N}$ is decisive w.r.t. $\mathbf{m}_1$.
Otherwise, considering the weights specified by W and adding loops for states without successors, this graph can be viewed as a finite Markov chain and so reaching some bottom strongly connected component (BSCC) almost surely. There are three possible cases: (1) the BSCC consisting of $\mathbf{m}_1$, (2) a BSCC consisting of a single marking $\mathbf{m}$ for which there exists some $p \in P$ fulfilling $\mathbf{m}_1(p) + B(\mathcal{N}, \mathbf{m}_0) < \mathbf{m}(p)$ and thus from which $\mathbf{m}_1$ is unreachable or (3) a BSCC that is also a BSCC of $\mathcal{M}_{\mathcal{N}}$ and thus from which one cannot reach $\mathbf{m}_1$. This establishes that $\mathcal{N}$ is decisive w.r.t. $\mathbf{m}_1$. ◀

In this particular case, instead of using Algorithm 1 to frame the reachability probability, one can use the Markov chain of the proof to exactly compute this probability.

6 Conclusion and perspectives

We have studied the decidability of decisiveness with respect to several subclasses of probabilistic counter machines. The results are summarized in the following table. When A is not mentioned it means that A is finite.

model	constant	polynomial	general
pHM	D	D [Th 14]	U [Th 13] even with a single state
pPN	?	U [Th 20] also w.r.t. upward closed sets[Th 20]	U but D when regular [Th 22]
pCM	U [Th 12]	U	U

In the future, apart for solving the left open problem in the above table, we plan to introduce sufficient conditions for decisiveness for models with undecidability of decisiveness like pPNs with polynomial weights. This could have a practical impact for real case-study modellings.

In another direction, we have established that the decisiveness and recurrence properties are closely related. It would be interesting to define a property related to transience in Markov chains. In fact we have identified such a property called divergence and the definition and analysis of this property will appear in a forthcoming paper.

References

1 Parosh Aziz Abdulla, Christel Baier, S. Purushothaman Iyer, and Bengt Jonsson. Reasoning about probabilistic lossy channel systems. In Catuscia Palamidessi, editor, *CONCUR 2000 - 11th International Conference on Concurrency Theory, University Park, PA, USA, August 22-25*, volume 1877 of *LNCS*, pages 320–333. Springer, 2000. doi:10.1007/3-540-44618-4_24.

2 Parosh Aziz Abdulla, Nathalie Bertrand, Alexander Moshe Rabinovich, and Philippe Schnoebelen. Verification of probabilistic systems with faulty communication. *Inf. Comput.*, 202(2):141–165, 2005. doi:10.1016/j.ic.2005.05.008.

3 Parosh Aziz Abdulla, Noomene Ben Henda, and Richard Mayr. Decisive Markov chains. *Log. Methods Comput. Sci.*, 3(4), 2007. doi:10.48550/arXiv.0706.2585.

4 Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996. doi:10.1109/LICS.1993.287591.

5 Parosh Aziz Abdulla and Alexander Moshe Rabinovich. Verification of probabilistic systems with faulty communication. In *Proceedings of the 6th international conference on Foundations of Software Science and Computational Structures FOSSACS*, volume 2620 of *LNCS*, pages 39–53. Springer, 2003. doi:10.1007/3-540-36576-1_3.

6 Christel Baier and Bettina Engelen. Establishing qualitative properties for probabilistic lossy channel systems: An algorithmic approach. In *Formal Methods for Real-Time and Probabilistic Systems, Proceedings of the 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999*, volume 1601 of *LNCS*, pages 34–52. Springer, 1999. doi:10.1007/3-540-48778-6_3.

7 Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

8 N. G. Bean, L. Bright, G. Latouche, C. E. M. Pearce, P. K. Pollett, and P. G. Taylor. The quasi-stationary behavior of quasi-birth-and-death processes. *The Annals of Applied Probability*, 7(1):134–155, 1997. doi:10.1214/aoap/1034625256.

9 Nathalie Bertrand, Patricia Bouyer, Thomas Brihaye, and Pierre Carlier. When are stochastic transition systems tameable? *J. Log. Algebraic Methods Program.*, 99:41–96, 2018. doi:10.48550/arXiv.1703.04806.

10 Nathalie Bertrand and Philippe Schnoebelen. Model checking lossy channels systems is probably decidable. In *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2003 Held as Part of ETAPS 2003, Warsaw, Poland, April 7-11*, volume 2620 of *LNCS*, pages 120–135. Springer, 2003. doi:10.1007/3-540-36576-1_8.

11 Tomás Brázdil, Krishnendu Chatterjee, Antonín Kucera, Petr Novotný, Dominik Velan, and Florian Zuleger. Efficient algorithms for asymptotic bounds on termination time in VASS. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 185–194. ACM, 2018. doi:10.1145/3209108.3209191.

12 Tomás Brázdil, Javier Esparza, Stefan Kiefer, and Antonín Kucera. Analyzing probabilistic pushdown automata. *Formal Methods Syst. Des.*, 43(2):124–163, 2013. doi:10.1007/s10703-012-0166-0.

13 Tomás Brázdil, Stefan Kiefer, and Antonín Kucera. Efficient analysis of probabilistic programs with an unbounded counter. *J. ACM*, 61(6):41:1–41:35, 2014. doi:10.1145/2629599.

14 Stéphane Demri. On selective unboundedness of VASS. *J. Comput. Syst. Sci.*, 79(5):689–713, 2013. doi:10.1016/j.jcss.2013.01.014.

15 Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming, ICALP'98*, volume 1443 of *LNCS*, pages 103–115. Springer, 1998. doi:10.5555/646252.686157.

16 Javier Esparza, Antonín Kucera, and Richard Mayr. Model Checking Probabilistic Pushdown Automata. *Logical Methods in Computer Science*, Volume 2, Issue 1, 2006. doi:10.2168/LMCS-2(1:2)2006.

17 Alain Finkel, Serge Haddad, and Lina Ye. About decisiveness of dynamic probabilistic models. *CoRR*, abs/2305.19564, 2023. doi:10.48550/arXiv.2305.19564.

18 Abraham Ginzburg and Michael Yoeli. Vector addition systems and regular languages. *J. Comput. Syst. Sci.*, 20(3):277–284, 1980. doi:10.1016/0022-0000(80)90009-4.

19 S. Purushothaman Iyer and Murali Narasimha. Probabilistic lossy channel systems. In *Theory and Practice of Software Development (TAPSOFT), 7th International Joint Conference CAAP/FASE*, volume 1214 of *LNCS*, pages 667–681. Springer, 1997. doi:10.1007/BFb0030633.

20 J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. Springer-Verlag, 2nd edition, 1976. doi:10.1007/978-1-4684-9455-6.

21 Tianrong Lin. Model-checking PCTL properties of stateless probabilistic pushdown systems with various extensions, 2023. doi:10.48550/arXiv.2209.10517.

22 Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., USA, 1967. doi:10.5555/1095587.

23 Alexander Moshe Rabinovich. Quantitative analysis of probabilistic lossy channel systems. In *Proceedings of 30th International Colloquium Automata, Languages and Programming ICALP*, volume 2719 of *LNCS*, pages 1008–1021. Springer, 2003. doi:10.1007/3-540-45061-0_78.

24 Eugene S. Santos. Probabilistic grammars and automata. *Inf. Control.*, 21(1):27–47, 1972. doi:10.1016/S0019-9958(72)90026-5.

25 Rüdiger Valk. Self-modifying nets, a natural extension of Petri nets. In *Proceedings of the 5th International Colloquium on Automata, Languages and Programming*, volume 62 of *LNCS*, pages 464–476. Springer, 1978. doi:10.1007/3-540-08860-1_35.

26 Rüdiger Valk and Guy Vidal-Naquet. Petri nets and regular languages. *J. Comput. Syst. Sci.*, 23(3):299–325, 1981. doi:10.1016/0022-0000(81)90067-2.

27 Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 327–338. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.12.

Probabilistic Operational Correspondence

Anna Schmitt
TU Darmstadt, Germany

Kirstin Peters
Augsburg University, Germany

Abstract

Encodings are the main way to compare process calculi. By applying quality criteria to encodings we analyse their quality and rule out trivial or meaningless encodings. Thereby, operational correspondence is one of the most common and most important quality criteria. It ensures that processes and their translations have the same abstract behaviour. We analyse probabilistic versions of operational correspondence to enable such a verification for probabilistic systems.

Concretely, we present three versions of probabilistic operational correspondence: weak, middle, and strong. We show the relevance of the weaker version using an encoding from a sublanguage of probabilistic CCS into the probabilistic π-calculus. Moreover, we map this version of probabilistic operational correspondence onto a probabilistic behavioural relation that directly relates source and target terms. Then we can analyse the quality of the criterion by analysing the relation it induces between a source term and its translation. For the second version of probabilistic operational correspondence we proceed in the opposite direction. We start with a standard simulation relation for probabilistic systems and map it onto a probabilistic operational correspondence criterion.

2012 ACM Subject Classification Theory of computation → Process calculi

Keywords and phrases Probabilistic Process Calculi, Encodings, Operational Correspondence

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.15

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2307.05218`

Acknowledgements We thank the anonymous reviewers of a preliminary version of this paper for their comments and hints.

1 Introduction

Encodings are used to compare process calculi and to reason about their expressive power. Encodability criteria are conditions that limit the existence of encodings. Their main purpose is to rule out trivial or meaningless encodings, but they can also be used to limit attention to encodings that are of special interest in a particular domain or for a particular purpose. These quality criteria are the main tool in separation results, saying that one calculus is not expressible in another one; here one has to show that no encoding meeting these criteria exists. To obtain stronger separation results, care has to be taken in selecting quality criteria that are not too restrictive. For encodability results, saying that one calculus is expressible in another one, all one needs is an encoding, together with criteria testifying for the quality of the encoding. Here it is important that the criteria are not too weak.

In the literature various different criteria and different variants of the same criteria are employed to achieve separation and encodability results (see e.g. [29] for an overview). Unfortunately it is not always obvious whether the criteria used to obtain a result in a particular setting do indeed fit to this setting. A way to formally analyse the quality of encodability criteria was presented in [30]. They propose to map the criteria on conditions on relations between source and target terms that in particular relate each source term with its literal translation. This allows us to formally reason about encodability criteria, to completely capture and describe their semantic effect, and to analyse side conditions of combinations of criteria. We want to use this technique to define and analyse versions of operational correspondence for probabilistic systems.

Intuitively, *operational correspondence* requires executions to be respected. It consists of a completeness and a soundness part. The completeness condition requires that for all source term executions there is one emulation in the target language such that the encoding of the result in the source and the result in the target are related by some relation $\mathcal{R}_{\mathsf{T}}$ on the target language. Intuitively, the completeness condition requires that any source term execution is emulated by the target term modulo $\mathcal{R}_{\mathsf{T}}$. Soundness requires that for all executions of the target there exists some execution of the source such that again the results are related by $\mathcal{R}_{\mathsf{T}}$. Intuitively, soundness requires that whatever the encoded term can do is a translation of some behaviour of the source term modulo $\mathcal{R}_{\mathsf{T}}$.

To study and compare probabilistic languages we need probabilistic versions of operational correspondence. In § 4 we start in the traditional way and use a particular encoding, that we consider reasonable. We derive a version of probabilistic operational correspondence (PrOC) that captures the way in that the encoding relates the behaviour of the source and the target language. Concretely, we consider an encoding from a sublanguage of probabilistic CCS from [7] into the probabilistic π-calculus of [38]. As result we obtain a weak version of PrOC.

In § 5 we then map the conditions in weak PrOC onto requirements of a relation between source and target terms as proposed in [30]. Thereby we are able to show that weak PrOC ensures that source terms and their literal translations are related by a probabilistic version of coupled similarity ([27]).

For our second variant of PrOC we are interested in a stricter criterion that induces a stricter simulation relation such as bisimulation. In § 6 we therefore reverse the ideas of [30]. Instead of starting with a criterion that we map on conditions of a relation, we start with an interesting relation and derive which variant of PrOC induces this relation. Since bisimulation is often considered as *the* standard behavioural relation between processes, we start with probabilistic barbed bisimulation. From this relation we derive a version of PrOC that induces this relation between source terms and their translations. Finally, we also present a strong version of PrOC and its correspondence to strong probabilistic bisimulation.

We conclude in § 7. The proofs and some additional material can be found in [33].

2 Process Calculi

A process calculus is a language $\mathcal{L} = (\mathcal{P}, \longmapsto)$ consisting of a set of terms $\mathcal{P}$ – its *syntax* – and its *semantics* defining reduction steps. The syntax of a process calculus is usually defined by a context-free grammar defining operators. An operator of arity 0 is a constant. The arguments that are again process terms are called *subterms*. A *guard* is an operator that prevents the reductions of subterms until the guard is reduced first. In the languages we consider, guards are action prefixes.

We assume that the semantics is given as an *operational semantics* consisting of inference rules defined on the operators of the language [31]. For many process calculi, the semantics is provided in two forms, as *reduction semantics* and as *labelled transition semantics*. We assume that at least the reduction semantics $\longmapsto$ is given as part of the definition, because its treatment is easier in the context of encodings. We consider probabilistic calculi, where a process reduces in a step to a discrete probability distribution.

A *(discrete) probability distribution* over a set S is a mapping $\Delta : S \to [0, 1]$ with $\sum_{P \in S} \Delta(P) = 1$. Let $\mathcal{D}(S)$ ranged over by Δ, Θ, Φ denote the collection of all such distributions over S. The *support* of Δ is the set $\lceil \Delta \rceil = \{P \mid \Delta(P) > 0\}$ of elements with a positive probability. We use $\overline{P}$ to denote the point distribution assigning probability 1 to state P and 0 to all other states in S. If $\sum_{i \in I} p_i = 1$, $p_i \geq 0$, and Δ_i is a probability distribution for each i in some finite index set I, then $\sum_{i \in I} p_i \cdot \Delta_i$ is a probability distribution given by $\left(\sum_{i \in I} p_i \cdot \Delta_i \right)(P) = \sum_{i \in I} p_i \cdot \Delta_i(P)$. We sometimes use $\{ p_1 P_1, \ldots, p_n P_n \}$ to denote a distribution Δ with $\lceil \Delta \rceil = \{ P_1, \ldots, P_n \}$ and $\Delta(P_i) = p_i$ for all $1 \leq i \leq n$.

A *(reduction) step* $P \longmapsto \Delta$ is a single application of the reduction relation $\longmapsto \; \subseteq \mathcal{P} \times \mathcal{D}(\mathcal{P})$, where Δ is called *derivative.* We lift $\longmapsto$ to a relation between distributions (see e.g. [7] for a similar but stricter relation).

▶ **Definition 1** (Reductions of Distributions). *Let $\Delta \longmapsto \Theta$ whenever*

(a) $\Delta = \sum_{i \in I} p_i \overline{P_i}$, *where I is a finite index set and* $\sum_{i \in I} p_i = 1$,

(b) *for each $i \in I$ there is a distribution Θ_i such that $P_i \longmapsto \Theta_i$ or $\Theta_i = \overline{P_i}$,*

(c) *for some $i \in I$ we have $P_i \longmapsto \Theta_i$, and*

(d) $\Theta = \sum_{i \in I} p_i \cdot \Theta_i$.

Let $P \longmapsto$ and $\Delta \longmapsto$ denote the existence of a single step from P or Δ. We write $P \longmapsto^\omega$ if there is some Δ such that $P \longmapsto \Delta$ and $\Delta \longmapsto^\omega$, where $\Delta \longmapsto^\omega$ if Δ has an *infinite sequence* of steps. Let $\Longmapsto$ be the reflexive and transitive closure of $\longmapsto$ on distributions and let $P \Longmapsto \Delta$ if $\Delta = \overline{P}$ or $P \longmapsto\Longmapsto \Delta$.

To reason about environments of terms, we use functions on process terms called contexts. More precisely, a *context* $\mathcal{C}([\cdot]_1, \ldots, [\cdot]_n) : \mathcal{P}^n \to \mathcal{P}$ with n holes is a function from n terms into one term, i.e., given $P_1, \ldots, P_n \in \mathcal{P}$, the term $\mathcal{C}(P_1, \ldots, P_n)$ is the result of inserting $P_1, \ldots, P_n$ in the corresponding order into the n holes of $\mathcal{C}$.

Assume a countably-infinite set $\mathcal{N}$ of *names.* Let $\overline{\mathcal{N}} = \{ \overline{n} \mid n \in \mathcal{N} \}$ and $\tau \notin \mathcal{N}$. Let $\mathsf{fn}(P)$ denote the set of free names in P. A substitution σ is a finite mapping from names to names defined by a set $\{ y_1/x_1, \ldots, y_n/x_n \} = \{ y_1, \ldots, y_n / x_1, \ldots, x_n \} = \{ \tilde{y}/\tilde{x} \}$ of renamings, where the $x_1, \ldots, x_n$ are pairwise distinct. The application $P\{ \tilde{y}/\tilde{x} \}$ of a substitution on a term is defined as the result of simultaneously replacing all free occurrences of x_i by y_i for $i \in \{ 1, \ldots, n \}$, possibly applying α-conversion to avoid capture or name clashes. For all names in $\mathcal{N} \setminus \{ x_1, \ldots, x_n \}$ the substitution behaves as the identity mapping. We naturally extend substitution to distributions.

For the last criterion of [15] (see Section 3), we need a special constant $\checkmark$, called *success(ful termination).* Therefore, we add $\checkmark$ to the grammar of a language without explicitly mentioning it. Success is used as a barb to implement some form of (fair) testing, where $P{\downarrow_\checkmark}$ if P has an unguarded occurrence of $\checkmark$, $\Delta{\downarrow_\checkmark}$ if there is some P with $\Delta(P) > 0$ and $P{\downarrow_\checkmark}$, and $P{\Downarrow_\checkmark} = \exists \Delta.\, P \Longmapsto \Delta \wedge \Delta{\downarrow_\checkmark}$.

Languages can be augmented with (a set of) relations $\mathcal{R} \subseteq \mathcal{P}^2$ on their processes. If $\mathcal{R} \subseteq B^2$ is a relation and $B' \subseteq B$, then the relation $\mathcal{R}\restriction_{B'} = \{ (x, y) \mid x, y \in B' \wedge (x, y) \in \mathcal{R} \}$ denotes the restriction of $\mathcal{R}$ to the domain B'.

Following [7], we lift relations $\mathcal{R} \subseteq \mathcal{P}^2$ to a relation $\overline{\mathcal{R}} \subseteq \mathcal{D}(\mathcal{P})^2$ on distributions.

▶ **Definition 2** (Relations on Distributions, [7]).
Let $\mathcal{R} \subseteq \mathcal{P}^2$ and let $\Delta, \Theta \in \mathcal{D}(\mathcal{P})$. Then $(\Delta, \Theta) \in \overline{\mathcal{R}}$ if

(a) $\Delta = \sum_{i \in I} p_i \overline{P_i}$, *where I is a finite index set and* $\sum_{i \in I} p_i = 1$,

(b) *for each $i \in I$ there is a process Q_i such that $(P_i, Q_i) \in \mathcal{R}$, and*

(c) $\Theta = \sum_{i \in I} p_i \overline{Q_i}$.

An important property of this lifting operation is that it preserves reflexivity and transitivity, i.e., if $\mathcal{R}$ is reflexive/transitive then so is $\overline{\mathcal{R}}$. The case of transitivity is shown in [7]. The case of reflexivity can be found in the technical report ([33]).

3 Encodings and Quality Criteria

Let $\mathcal{L}_{\mathsf{S}} = \langle \mathcal{P}_{\mathsf{S}}, \longmapsto_{\mathsf{S}} \rangle$ and $\mathcal{L}_{\mathsf{T}} = \langle \mathcal{P}_{\mathsf{T}}, \longmapsto_{\mathsf{T}} \rangle$ be two process calculi, denoted as *source* and *target* language. An *encoding* from $\mathcal{L}_{\mathsf{S}}$ into $\mathcal{L}_{\mathsf{T}}$ is a function $[\![\cdot]\!] : \mathcal{P}_{\mathsf{S}} \to \mathcal{P}_{\mathsf{T}}$. We often use $S, S', \ldots$ and $T, T', \ldots$ to range over $\mathcal{P}_{\mathsf{S}}$ and $\mathcal{P}_{\mathsf{T}}$, respectively.

We naturally extend the encoding function to distributions, i.e., for all distributions Δ in the source language and all T in the target language: $[\![\Delta]\!](T) = \sum_{S \in \{S \in \mathcal{P}_S \mid [\![S]\!] = T\}} \Delta(S)$.

Let $\varphi_{[\![\cdot]\!]} : \mathcal{N} \to \mathcal{N}^k$ be a *renaming policy*, i.e., a mapping from a name to a vector of names that can be used by encodings to split names and to reserve special names, such that no two different names are translated into overlapping vectors of names. We use projection to obtain the respective elements of a translated name, i.e., if $\varphi_{[\![\cdot]\!]}(a) = (a_1, a_2, a_3)$ then $\varphi_{[\![\cdot]\!]}(a).2 = a_2$. Slightly abusing notation, we sometimes use the tuples that are generated by the renaming policy as sets. We require e.g. $\varphi_{[\![\cdot]\!]}(a) \cap \varphi_{[\![\cdot]\!]}(b) = \emptyset$ whenever $a \neq b$.

To analyse the quality of encodings and to rule out trivial or meaningless encodings, they are augmented with a set of quality criteria. One such set of criteria that is well suited for separation as well as encodability results between traditional processes calculi, i.e., calculi without probabilities, was proposed in [15]. It turns out that for probabilistic systems as defined above the only criterion that needs to be adapted is operational correspondence. Accordingly, we inherit the remaining criteria from [15]:

Compositionality: For every operator $\mathbf{op}$ with arity n of $\mathcal{L}_{\mathsf{S}}$ and for every subset of names N, there exists a context $\mathcal{C}_{\mathbf{op}}^{N}([\cdot]_1, \ldots, [\cdot]_n)$ such that, for all $S_1, \ldots, S_n$ with $\mathsf{fn}(S_1) \cup \ldots \cup \mathsf{fn}(S_n) = N$, it holds that $[\![\mathbf{op}\,(S_1, \ldots, S_n)]\!] = \mathcal{C}_{\mathbf{op}}^{N}([\![S_1]\!], \ldots, [\![S_n]\!])$.

Name Invariance w.r.t. a Relation $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^{2}$: For every $S \in \mathcal{P}_{\mathsf{S}}$ and every substitution σ, it holds that $[\![S\sigma]\!] \equiv_\alpha [\![S]\!]\sigma'$ if σ is injective and $([\![S\sigma]\!], [\![S]\!]\sigma') \in \mathcal{R}_{\mathsf{T}}$ otherwise, where σ' is such that $\varphi_{[\![\cdot]\!]}(\sigma(a)) = \sigma'\big(\varphi_{[\![\cdot]\!]}(a)\big)$ for all $a \in \mathcal{N}$.

Divergence Reflection: For every S, $[\![S]\!] \longmapsto^{\omega}$ implies $S \longmapsto^{\omega}$.

Success Sensitiveness: For every S, $S{\Downarrow_{\checkmark}}$ iff $[\![S]\!]{\Downarrow_{\checkmark}}$.

Compositionality ensures that encodings are of practical relevance by enforcing that they can be implemented compositionally, i.e., by an algorithm that proceeds on the syntax and does not need to analyse the source term to compute its translation. However, the formulation of compositionality is rather strict, i.e., it rules out practically relevant translations. Note that the best known encoding from the asynchronous π-calculus into the Join Calculus in [11] is not compositional, but consists of an inner, compositional encoding surrounded by a fixed context – the implementation of so-called firewalls – that is parameterised on the free names of the source term. In order to capture this and similar encodings we relax the definition of compositionality.

Weak Compositionality: The encoding is either compositional or consists of an inner, compositional encoding surrounded by a fixed context that can be parameterised on the free names of the source term or information that are not part of the source term.

Precisely, we use this relaxation to capture process definitions that are a relevant part of the source language $\mathsf{CCS}_{\mathsf{p}}$ but that are not contained in source terms.

A behavioural relation $\mathcal{R}_{\mathsf{T}}$ on the target is assumed for name invariance and operational correspondence. $\mathcal{R}_{\mathsf{T}}$ needs to be success sensitive, i.e., $(T_1, T_2) \in \mathcal{R}_{\mathsf{T}}$ implies $T_1{\Downarrow_{\checkmark}}$ iff $T_2{\Downarrow_{\checkmark}}$.

Operational correspondence is arguably the most important of the five criteria in [15], since it compares the behaviour of source terms and their translations (though only the combination with success sensitiveness ensures that this requirement is not trivial). It consists of a soundness and a completeness condition. *Completeness* requires that every computation of a source term can be emulated by its translation. *Soundness* requires that every computation of a target term corresponds to some computation of the corresponding source term. Different variants of operational correspondence are used in the literature (see e.g. [30, 29]). In particular, the following three variants are often used for encodings between process calculi without probabilities.

▶ **Definition 3** (Operational Correspondence, Non-Probabilistic).
An encoding $\llbracket \cdot \rrbracket$ *is* strongly operationally corresponding *w.r.t.* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *if it is:*

Strongly Complete: $\forall S, S'.\ S \longmapsto S'$ *implies* $(\exists T.\ \llbracket S \rrbracket \longmapsto T \wedge (\llbracket S' \rrbracket, T) \in \mathcal{R}_{\mathsf{T}})$

Strongly Sound: $\forall S, T.\ \llbracket S \rrbracket \longmapsto T$ *implies* $(\exists S'.\ S \longmapsto S' \wedge (\llbracket S' \rrbracket, T) \in \mathcal{R}_{\mathsf{T}})$

$\llbracket \cdot \rrbracket$ *is* operationally corresponding *w.r.t.* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *if it is:*

Complete: $\forall S, S'.\ S \Longmapsto S'$ *implies* $(\exists T.\ \llbracket S \rrbracket \Longmapsto T \wedge (\llbracket S' \rrbracket, T) \in \mathcal{R}_{\mathsf{T}})$

Sound: $\forall S, T.\ \llbracket S \rrbracket \Longmapsto T$ *implies* $(\exists S'.\ S \Longmapsto S' \wedge (\llbracket S' \rrbracket, T) \in \mathcal{R}_{\mathsf{T}})$

$\llbracket \cdot \rrbracket$ *is* weakly operationally corresponding *w.r.t.* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *if it is:*

Complete: $\forall S, S'.\ S \Longmapsto S'$ *implies* $(\exists T.\ \llbracket S \rrbracket \Longmapsto T \wedge (\llbracket S' \rrbracket, T) \in \mathcal{R}_{\mathsf{T}})$

Weakly Sound: $\forall S, T.\ \llbracket S \rrbracket \Longmapsto T$ *impl.* $(\exists S', T'.\ S \Longmapsto S' \wedge T \Longmapsto T' \wedge (\llbracket S' \rrbracket, T') \in \mathcal{R}_{\mathsf{T}})$

$$\begin{tikzcd}
S \arrow[rr, Rightarrow, maps to] \arrow[dd, dotted, "\llbracket \cdot \rrbracket"] & & S' \arrow[d, dotted, "\llbracket \cdot \rrbracket"] \\
 & & \llbracket S' \rrbracket \arrow[d, dash, dashed, "\mathcal{R}_{\mathsf{T}}"] \\
\llbracket S \rrbracket \arrow[r, Rightarrow, maps to] & T \arrow[r, Rightarrow, maps to] & T'
\end{tikzcd}$$

The first variant requires a strong correspondence between source and target term steps, i.e., a source term step is emulated by exactly one target term step and vice versa. The second variant allows for the emulation of a single source term step by a sequence of target term steps. With this variant encoding functions may use things like pre- and post-processing steps. The last variant additionally allows for *intermediate states*, i.e., target terms that are not directly related to the encoding of any source term but that are in between two such source term encodings [27, 28, 17]. Such an intermediate state is depicted by T on the right. Intermediate states often result from partial commitments: The decision on which step is performed for a single source term step is split into two or more partial decisions in a sequence of target term steps, where each decision already rules out some of the alternatives that existed in the source for this step but does not rule out all alternatives.

4 PrOC for a Reasonable Encoding

The quality criteria of encodings should rule out trivial and meaningless encodings but they should capture good encodings. Accordingly, we start with an intuitively reasonable encoding (from $\mathsf{CCS}_{\mathsf{p}}$ into π_{p}) and derive a version of PrOC that captures how this encoding translates the behaviour of source terms. Then we analyse the quality of our new criterium by mapping it on a relation between source and target terms in Section 5.

Our source language, probabilistic CCS, is introduced in [7] as a probabilistic extension of CCS [21]. We omit the operator for non-deterministic choice from [7], because its summands are not necessarily guarded, whereas our target language has only guarded choice. Thus, ignoring this operator simplifies the task of finding an encoding. Since for the design of encodability criteria the consideration of non-determinism and unguarded choices is orthogonal to the consideration of probabilities, it is safe to neglect this operator here. The versions of PrOC we discover in the following can deal with combinations of non-determinism and probabilities. We denote the resulting calculus as $\mathsf{CCS_p}$. Let $u, v, \ldots$ range over the set of *actions* $\mathsf{Act} = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$.

▶ **Definition 4** (Syntax of $\mathsf{CCS_p}$)**.** *The terms $\mathcal{P}_C$ of $\mathsf{CCS_p}$ are given by:*

$$P \ ::= \ u.\bigoplus_{i\in I} p_i P_i \quad | \quad P_1 \mid P_2 \quad | \quad P \setminus A \quad | \quad P[f] \quad | \quad C\langle\tilde{x}\rangle$$

where $A \subseteq \mathcal{N}$ and $f : \mathcal{N} \to \mathcal{N}$ is a renaming function.

The probabilistic choice operator $u.\bigoplus_{i\in I} p_i P_i$ is guarded by an action u and offers branches with probabilities, where $p_i > 0$ is the probability of branch i with $\sum_{i\in I} p_i = 1$. We write $\bigoplus_{i\in 1..n} p_i P_i$ as $p_1 P_1 \oplus \ldots \oplus p_n P_n$ for a finite index set I. The process $P_1 \mid P_2$ implements parallel composition, in $P \setminus A$ the names in A are restricted, and $P[f]$ behaves like P where each $a \in \mathcal{N}$ is replaced by $f(a)$. For simplicity, we assume that for all f the set $\{x \mid f(x) \neq x\}$ is finite. Each process constant C has a definition $C \overset{\text{def}}{=} (\tilde{x})P$, where $P \in \mathcal{P}_C$ and $\tilde{x}$ collect all names in P that are not restricted. Then $C\langle\tilde{y}\rangle$ behaves as P with $\tilde{y}$ replacing $\tilde{x}$.

As described in [33], we use a reduction semantics obtained from the labelled semantics in [7] by a rule that maps every τ-labelled step to a reduction step. Due to lack of space, we only highlight the rules for choice and communication here and refer to [33] for the rest.

$$\textsc{ProbChoice}_{\mathsf{CCS_p}} \ \frac{\Delta(P) = \sum\{p_i \mid i \in I \wedge P_i = P\}}{u.\bigoplus_{i\in I} p_i P_i \xrightarrow{u} \Delta} \qquad \textsc{ComL}_{\mathsf{CCS_p}} \ \frac{P_1 \xrightarrow{a} \Delta_1 \quad P_2 \xrightarrow{\overline{a}} \Delta_2}{P_1 \mid P_2 \xrightarrow{\tau} \Delta_1 \mid \Delta_2}$$

where $(\Delta_1 \mid \Delta_2)(P) = \begin{cases} \Delta_1(P_1) \cdot \Delta_2(P_2) & \text{, if } P = P_1 \mid P_2 \\ 0 & \text{otherwise} \end{cases}$.

Our target language, the probabilistic π-calculus (π_{p}), is introduced in [38], as a probabilistic version of the πI-calculus [32], where output is endowed with probabilities.

▶ **Definition 5** (Syntax of π_{p})**.** *The terms $\mathcal{P}_\pi$ of π_{p} are given by:*

$$P \ ::= \ \overline{x} \oplus_{i\in I} p_i\, \boldsymbol{in}_i(\tilde{y}_i).P_i \quad | \quad x\Phi_{i\in I}\, \boldsymbol{in}_i(\tilde{y}_i).P_i \quad | \quad P \mid P \quad | \quad (\nu x)P \quad | \quad \mathbf{0} \quad | \quad !x(\tilde{y}).P$$

The probabilistic π-calculus assigns probabilities to output. The process $\overline{x} \oplus_{i\in I} p_i \mathtt{in}_i(\tilde{y}_i).P_i$ is a probabilistic selecting output, where $p_i \in [0,1]$ for all $i \in I$ and $\sum_{i\in I} p_i = 1$. The term $x\Phi_{i\in I}\mathtt{in}_i(\tilde{y}_i).P_i$ is a branching input, which does not attach probabilities to the single events. For branching/selection labels, the index i is the branch of the label. We write $\overline{x}(\tilde{y}).P$ and $x(\tilde{y}).P$ for single outputs or inputs and $\overline{x}\left(p_1\mathtt{in}_1\left(\tilde{y_1}\right).P_1 \oplus \ldots \oplus p_n\mathtt{in}_n\left(\tilde{y_n}\right).P_n\right)$ and $x\left(\mathtt{in}_1\left(\tilde{y_1}\right).P_1 \,\&\ldots\&\, \mathtt{in}_n\left(\tilde{y_n}\right).P_n\right)$ for finite indexing sets $I = \{1, \ldots, n\}$ in $\oplus_{i\in I}$ and $\Phi_{i\in I}$. The process $P \mid Q$ is a parallel composition, $(\nu x)P$ is a restriction, and $!x(\tilde{y}).P$ is a replicated input. We sometimes omit empty sequences of arguments () as well as dangling $\mathbf{0}$, i.e., $\overline{a}$ stands for $\overline{a}().\mathbf{0}$.

Structural congruence $\equiv$ is defined, similarly to [22], as the smallest congruence containing α-equivalence $\equiv_\alpha$ that is closed under the following rules:

$$\begin{gathered} P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\nu x)\mathbf{0} \equiv \mathbf{0} \\ (\nu xy)P \equiv (\nu yx)P \qquad (\nu x)(P \mid Q) \equiv P \mid (\nu x)Q \quad \text{if } x \notin \mathsf{fn}(P) \end{gathered}$$

We lift structural congruence to distributions, i.e., $\Delta_1 \equiv \Delta_2$ if there is a finite index set I such that $\Delta_1 = \sum_{i\in I} p_i \overline{P_i}$, $\Delta_2 = \sum_{i\in I} p_i \overline{Q_i}$, and $P_i \equiv Q_i$ for all $i \in I$.

$$\begin{array}{lcl}
\left\llbracket x.\bigoplus_{i\in I} p_i P_i \right\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & x.(\nu z_i)\left(\overline{z_i} \oplus_{i\in I} p_i \mathtt{in}_i.\,\llbracket P_i\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} \mid z_i\right)\\
\left\llbracket \overline{x}.\bigoplus_{i\in I} p_i P_i \right\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & \overline{x} \oplus_{i\in I} p_i \mathtt{in}_i.\,\llbracket P_i\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}\\
\left\llbracket \tau.\bigoplus_{i\in I} p_i P_i \right\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & (\nu z_\tau)\left(\overline{z_\tau} \oplus_{i\in I} p_i \mathtt{in}_i.\,\llbracket P_i\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} \mid z_\tau\right)\\
\llbracket P \mid Q\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & \llbracket P\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} \mid \llbracket Q\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}\\
\llbracket P \setminus A\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & (\nu A)\,\llbracket P\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}\\
\llbracket P[f]\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & \llbracket P\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}\,\{\mathsf{ran}_f/\mathsf{dom}_f\}\\
\llbracket C\langle \tilde{y}\rangle\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & \overline{C}(\tilde{y})\\
\llbracket \checkmark\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} & = & \checkmark
\end{array}$$

where for each f the $\mathsf{ran}_f = y_1, \ldots, y_n$ and $\mathsf{dom}_f = x_1, \ldots, x_n$ are vectors of names such that $\{x_1, \ldots, x_n\} = \{x \mid f(x) \neq x\}$ and $f(x_i) = y_i$ for all $1 \leq i \leq n$.

Figure 1 Inner Encoding.

Again we refer to [33] or [38] for the semantics of π_p and highlight the rule for selection.

$$\textsc{Select}_{\pi_\mathsf{p}}\ \overline{x} \oplus_{i\in I} p_i \mathtt{in}_i(\tilde{y}_i).P_i \left\{ \xrightarrow[p_i]{\overline{x}\mathtt{in}_i\langle \tilde{y}_i\rangle} P_i \right\}_{i\in I}$$

We observe that in [38] the semantics is given as a Segala automaton. To obtain a reduction semantics with steps into probability distributions, we add the following rule:

$$\textsc{Red}_{\pi_\mathsf{p}}\ \frac{P\left\{ \xrightarrow[p_i]{\tau} Q_i \right\}_{i\in I} \qquad \Delta(R) = \sum\{p_i \mid Q_i = R\}}{P \longmapsto \Delta}$$

An encoding from $\mathsf{CCS_p}$ into π_p has to deal with the following challenge: Probabilistic choice in $\mathsf{CCS_p}$ can have output, input, and τ guards, but in π_p all summands of a probabilistic choice have to be output guarded. The input guarded choice operator in π_p has no probabilities and the τ is not part of the syntax. Therefore, the encoding of a probabilistic choice in $\mathsf{CCS_p}$ is split into three cases depending on its guard and the probabilistic selecting output of the target language is used in each of these cases to assign the probabilities.

We use the renaming policy $\varphi_{(\!|\cdot|\!)^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}}$ to reserve the names z_i (for input guarded probabilistic choice) and z_τ (for τ-guarded probabilistic choice). Moreover, we translate process constants C into channel names C and use the renaming policy to keep these channel names C distinct from source term names. Precisely, we assume that $|\varphi_{(\!|\cdot|\!)^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}}(n)| = 1$ for all $n \in \mathcal{N}$ and that $\varphi_{(\!|\cdot|\!)^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}}(n) \cap \{z_i, z_\tau, C \mid C \text{ is a process constant}\} = \emptyset$ for all $n \in \mathcal{N}$. In the following, in order to increase readability, the indication of the renaming policy is omitted, i.e., we assume $z_i \neq n \neq z_\tau$ and $n \neq C$ for all source term names n and all process constants C and write n instead of $\varphi_{(\!|\cdot|\!)^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}}(n).1$ in the translation.

▶ **Definition 6** (Encoding $(\!|\cdot|\!)^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}/\llbracket\cdot\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}$ from $\mathsf{CCS_p}$ into π_p). *The encoding of $S \in \mathcal{P}_C$ with the process definitions $C_1 \overset{def}{=} (\tilde{x}_1).S_1, \ldots, C_n \overset{def}{=} (\tilde{x}_n).S_n$ consists of the outer encoding $(\!|\cdot|\!)^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}$, where $(\!|S|\!)^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}$ is*

$$(\nu C_1, \ldots, C_n)\left(\llbracket S\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} \mid !C_1(\tilde{x}_1).\,\llbracket S_1\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}} \mid \ldots \mid !C_n(\tilde{x}_n).\,\llbracket S_n\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}\right)$$

and the inner encoding $\llbracket\cdot\rrbracket^{\pi_\mathsf{p}}_{\mathsf{CCS_p}}$ is given in Figure 1.

The encoding of a probabilistic choice is split into three cases. For input guards a single input on x is used, to enable the communication with a potential corresponding output. Then a probabilistic selecting output on the reserved name z_i composed in parallel with a matching input is used to encode the probabilities. The sequence of these two communication steps on x and z_i emulates the behaviour of a single communication step in the source.

The encoding of an output-guarded probabilistic choice is straightforward, as it is translated using the probabilistic selecting output.

For the guard τ, an output-guarded probabilistic choice in parallel to a single input on the reserved name z_τ is used.

The application of a renaming function is encoded by a substitution. A call $C\langle\tilde{y}\rangle$ is encoded by an output, where the corresponding process definitions are translated into replicated inputs and placed in parallel by the outer encoding. The remaining translations are homomorphic.

We are looking for a variant of operational correspondence with probabilities that captures the way in that our encoding emulates source term steps. By Definition 6, the emulation of a communication step consists of a sequence containing two target term steps. Accordingly, we are looking for a variant of operational correspondence that permits a sequence of target term steps to emulate a single source term step as in the second and third case of Definition 3.

▶ **Example 7.** Consider $S = \overline{x}.\left(\frac{3}{4}P \oplus \frac{1}{4}Q\right) \mid x.\left(\frac{1}{2}R \oplus \frac{1}{2}S\right)$ in $\mathsf{CCS_p}$ without process definitions. By the semantics of $\mathsf{CCS_p}$, $S \longmapsto \Delta_S = \left\{\frac{3}{8}(P \mid R), \frac{3}{8}(P \mid S), \frac{1}{8}(Q \mid R), \frac{1}{8}(Q \mid S)\right\}$. By Definition 6, $\llparenthesis S \rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} = \llbracket S \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ and:

$$\begin{aligned}\llbracket S \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} = \; & \overline{x}\left(\tfrac{3}{4}\mathtt{in}_1.\,\llbracket P \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \oplus \tfrac{1}{4}\mathtt{in}_2.\,\llbracket Q \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}\right) \mid \\ & x.(\nu z_i)\Big(\overline{z_i}\left(\tfrac{1}{2}\mathtt{in}_1.\,\llbracket R \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \oplus \tfrac{1}{2}\mathtt{in}_2.\,\llbracket S \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}\right) \mid z_i\Big)\end{aligned}$$

By the semantics of π_{p}, $\llparenthesis S \rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ can perform exactly one maximal sequence of steps, namely $\llparenthesis S \rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \longmapsto \Delta_T \longmapsto \Delta_T'$, where:

$$\begin{aligned}\Delta_T = \Big\{ \; & \tfrac{3}{4}\Big(\llbracket P \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid (\nu z_i)\Big(\overline{z_i}\left(\tfrac{1}{2}\mathtt{in}_1.\,\llbracket R \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \oplus \tfrac{1}{2}\mathtt{in}_2.\,\llbracket S \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}\right) \mid z_i\Big)\Big), \\ & \tfrac{1}{4}\Big(\llbracket Q \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid (\nu z_i)\Big(\overline{z_i}\left(\tfrac{1}{2}\mathtt{in}_1.\,\llbracket R \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \oplus \tfrac{1}{2}\mathtt{in}_2.\,\llbracket S \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}\right) \mid z_i\Big)\Big)\Big\} \\ \Delta_T' = \Big\{ \; & \tfrac{3}{8}\Big(\llbracket P \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \llbracket R \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \mathbf{0}\Big), \tfrac{3}{8}\Big(\llbracket P \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \llbracket S \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \mathbf{0}\Big), \\ & \tfrac{1}{8}\Big(\llbracket Q \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \llbracket R \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \mathbf{0}\Big), \tfrac{1}{8}\Big(\llbracket Q \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \llbracket S \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \mid \mathbf{0}\Big)\Big\}\end{aligned}$$

We observe that $\llparenthesis \Delta_S \rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \equiv \Delta_T'$ and in particular that the distributions Δ_S and Δ_T' have the same probabilities. However, the distribution Δ_T is not that obviously related to $\llparenthesis S \rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ or $\llparenthesis \Delta_S \rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$. ◀

We already ruled out strong operational correspondence as defined in Definition 3. The other two versions differ in whether they allow for intermediate states. Another look at Example 7 tells us that intermediate states make sense. Δ_T is a finite probability distribution with two cases: the case containing $\llbracket P \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ with probability $\frac{3}{4}$ and the case containing $\llbracket Q \rrbracket^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ with probability $\frac{1}{4}$, but neither S nor Δ_S have cases with these probabilities. In the second variant of operational correspondence in Definition 3 without intermediate states, we would need to find a relation $\mathcal{R}_{\mathsf{T}}$ that relates Δ_T either to $\llparenthesis S \rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ or Δ_T'. Such a relation $\mathcal{R}_{\mathsf{T}}$ is difficult or at least not intuitive, since it has to relate states with different probabilities. It is easier to allow for intermediate states. So, we want to build a weak version of operational correspondence (third case of Definition 3) with probabilities.

As discussed above, we need to require that the resulting distribution in the source has the same probabilities as the resulting distribution in the target term sequence. Moreover, it makes sense to require that for all matching cases, i.e., all branches with the same probability, the encoding of the respective source term and the respective target term are related by $\mathcal{R}_{\mathsf{T}}$. Definition 2 allows us to formalise this by comparing the distributions with $\overline{\mathcal{R}_{\mathsf{T}}}$. This leads to the version of probabilistic operational correspondence below denoted as *weak probabilistic operational correspondence.*

▶ **Definition 8** (Weak Probabilistic Operational Correspondence)**.** *An encoding* $[\![\cdot]\!] : \mathcal{P}_{\mathsf{S}} \to \mathcal{P}_{\mathsf{T}}$ *is* weakly probabilistic operationally corresponding *(weak PrOC) w.r.t.* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *if it is:*

Probabilistic Complete:

$$\forall S, \Delta_S .\ S \Longmapsto \Delta_S \textit{ implies } \left(\exists \Delta_T .\ [\![S]\!] \Longmapsto \Delta_T \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\right)$$

Weakly Probabilistic Sound: $\forall S, \Delta_T .\ [\![S]\!] \Longmapsto \Delta_T$ *implies*

$$\left(\exists \Delta_S, \Delta_T' .\ S \Longmapsto \Delta_S \wedge \Delta_T \Longmapsto \Delta_T' \wedge ([\![\Delta_S]\!], \Delta_T') \in \overline{\mathcal{R}_{\mathsf{T}}}\right)$$

Every source term step is emulated by one or two target term steps modulo $\equiv$ (the standard structural congruence on the target language). We prove in [33] that this holds for all source term steps and that the encoding satisfies weak PrOC. Weak compositionality holds by definition. Name invariance follows from the strict use of the renaming policy and because the encoding does not introduce free names. Divergence reflection results from weak probabilistic operational soundness. Finally, weak probabilistic operational correspondence and the homomorphic translation of success ensure that $(\!|\cdot|\!)_{\mathsf{CCS_p}}^{\pi_{\mathsf{p}}}$ is success sensitive.

▶ **Theorem 9.** *The encoding* $(\!|\cdot|\!)_{\mathsf{CCS_p}}^{\pi_{\mathsf{p}}}$ *satisfies weak compositionality, name invariance, weak probabilistic operational correspondence w.r.t.* $\equiv$*, divergence reflection, and success sensitiveness.*

To ensure that weak PrOC is a meaningful criterion, we use the technique for analysing encodability criteria presented in [30]. Therefore, weak PrOC is mapped on requirements on a relation between source and target terms.

5 Analysing Weak Probabilistic Operational Correspondence

To analyse the quality of the criterion weak PrOC in Definition 8, we follow the technique presented in [30] and map this criterion on requirements of a behavioural relation between source and target. Thus, an encoding that satisfies this criterion relates source terms and their translations in the target modulo the respective behavioural relation. This transfers the task to analyse the quality of weak PrOC to the quality of the behavioural relation it is mapped on. As such relations are a well researched area, this allows us to formally reason about the underlying version of operational correspondence.

In [30] the relation between different versions of operational correspondence and behavioural relations is shown. Therefore, requirements are derived such that an encoding is operational corresponding w.r.t. the considered variant of operational correspondence iff a behavioural relation with these requirements exists that relates source terms with their translations. The derived relation describes how similar the behaviour of the source term is to its translation and, thus, tells us about the quality of the criterion.

Among the versions of operational correspondence considered in [30] weak PrOC is closest to weak operational correspondence in Definition 3. The corresponding result relates weak operational correspondence and correspondence similarity.

▶ **Lemma 10** (Weak Operational Correspondence, [30]). $[\![\cdot]\!]$ *is weakly operationally corresponding w.r.t. a preorder* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *that is a correspondence simulation iff* $\exists \mathcal{R}_{[\![\cdot]\!]}.\ \big(\forall S.\ (S, [\![S]\!]) \in \mathcal{R}_{[\![\cdot]\!]}\big) \wedge \mathcal{R}_{\mathsf{T}} = \mathcal{R}_{[\![\cdot]\!]}\!\restriction_{\mathcal{P}_{\mathsf{T}}} \wedge \big(\forall S, T.\ (S, T) \in \mathcal{R}_{[\![\cdot]\!]} \rightarrow ([\![S]\!], T) \in \mathcal{R}_{\mathsf{T}}\big) \wedge \mathcal{R}_{[\![\cdot]\!]}$ *is a preorder and a correspondence simulation.*

Remember that a preorder is a binary relation that is reflexive and transitive. The relation $\mathcal{R}_{[\![\cdot]\!]}$ (here and in all of the following results) is a set of pairs over the disjoint union of source and target terms. *Correspondence similarity* is a simulation relation in between bisimilarity and coupled similarity, that was derived in [30] to exactly capture the nature of weak operational correspondence.

▶ **Definition 11** (Correspondence Simulation, [30]). *A relation* $\mathcal{R}$ *is a* (weak reduction) correspondence simulation *if for each* $(P, Q) \in \mathcal{R}$*:*

- $P \Longmapsto P'$ *implies* $\exists Q'.\ Q \Longmapsto Q' \wedge (P', Q') \in \mathcal{R}$
- $Q \Longmapsto Q'$ *implies* $\exists P'', Q''.\ P \Longmapsto P'' \wedge Q' \Longmapsto Q'' \wedge (P'', Q'') \in \mathcal{R}$

Two terms are correspondence similar *if a correspondence simulation relates them.*

To obtain a result similar to Lemma 10 for weak PrOC, we use Definition 2 to lift the definition of correspondence similarity to probability distributions.

▶ **Definition 12** (Probabilistic Correspondence Simulation). *A relation* $\mathcal{R}$ *is a* (weak) probabilistic (reduction) correspondence simulation *if for each* $(P, Q) \in \mathcal{R}$*:*

- $P \Longmapsto \Delta$ *implies* $\exists \Theta.\ Q \Longmapsto \Theta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$
- $Q \Longmapsto \Theta$ *implies* $\exists \Delta', \Theta'.\ P \Longmapsto \Delta' \wedge \Theta \Longmapsto \Theta' \wedge (\Delta', \Theta') \in \overline{\mathcal{R}}$

Two terms are probabilistic correspondence similar *if a probabilistic correspondence simulation relates them.*

To use probabilistic correspondence similarity, we have to show that the lifting operation in Definition 2 preserves the property of being a probabilistic correspondence simulation (at least for preorders). A relation $\overline{\mathcal{R}}$ on distributions is a probabilistic correspondence simulation if it satisfies Definition 11 with distributions instead of processes.

▶ **Lemma 13** (Preservation of the Correspondence Property).
If the preorder $\mathcal{R}$ *is a probabilistic correspondence simulation then so is* $\overline{\mathcal{R}}$*.*

With the probabilistic version of correspondence similarity we can adapt Lemma 10 to weak PrOC.

▶ **Theorem 14** (Weak PrOC). $[\![\cdot]\!]$ *is weakly probabilistically operationally corresponding w.r.t. a preorder* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *that is a probabilistic correspondence simulation iff* $\exists \mathcal{R}_{[\![\cdot]\!]}.\ \big(\forall S.\ (S, [\![S]\!]) \in \mathcal{R}_{[\![\cdot]\!]}\big) \wedge \mathcal{R}_{\mathsf{T}} = \mathcal{R}_{[\![\cdot]\!]}\!\restriction_{\mathcal{P}_{\mathsf{T}}} \wedge \big(\forall S, T.\ (S, T) \in \mathcal{R}_{[\![\cdot]\!]} \longrightarrow ([\![S]\!], T) \in \mathcal{R}_{\mathsf{T}}\big) \wedge \mathcal{R}_{[\![\cdot]\!]}$ *is a preorder and a probabilistic correspondence simulation.*

To prove this theorem, we have to construct (for the if-case) a relation $\mathcal{R}_{[\![\cdot]\!]}$ from $\mathcal{R}_{\mathsf{T}}$. Therefore, we use $\mathsf{t}(\mathsf{r}(\mathcal{R}_{\mathsf{T}} \cup \{(S, [\![S]\!]) \mid S \in \mathcal{P}_{\mathsf{S}}\}))$, where t and r denote transitive and reflexive closure. Then we show that in both directions the respective properties imply each other. In particular, we establish the relation between weak PrOC and the definition of probabilistic correspondence simulation, i.e., completeness of weak PrOC in Definition 8

$$\forall S, \Delta_S.\ S \Longmapsto \Delta_S \longrightarrow \big(\exists \Delta_T.\ [\![S]\!] \Longmapsto \Delta_T \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\big)$$

is mapped on the first condition of Definition 12

$$P \Longmapsto \Delta \longrightarrow \big(\exists \Theta.\ Q \Longmapsto \Theta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}\big)$$

and weak soundness of weak PrOC

$$\forall S, \Delta_T.\ [\![S]\!] \Longmapsto \Delta_T \longrightarrow \left(\exists \Delta_S, \Delta_T'.\ S \Longmapsto \Delta_S \wedge \Delta_T \Longmapsto \Delta_T' \wedge ([\![\Delta_S]\!], \Delta_T') \in \overline{\mathcal{R}_{\mathsf{T}}}\right)$$

is mapped on the seconded condition of Definition 12:

$$Q \Longmapsto \Theta \longrightarrow \left(\exists \Delta', \Theta'.\ P \Longmapsto \Delta' \wedge \Theta \Longmapsto \Theta' \wedge (\Delta', \Theta') \in \overline{\mathcal{R}}\right)$$

The condition $\forall S, T.\ (S, T) \in \mathcal{R}_{[\![\cdot]\!]} \longrightarrow ([\![S]\!], T) \in \mathcal{R}_{\mathsf{T}}$ is necessary to ensure (with the remaining properties) that the encoding satisfies weak PrOC in the “only if”-case of Theorem 14. Although the formulation of weak PrOC and probabilistic correspondence simulation are quite close, to prove that they are related is technically challenging. We have to show that the properties of the involved relations are preserved when we lift the relations on distributions with Definition 2 and that the probabilistic version of the correspondence similarity exactly captures the probabilistic nature of weak PrOC.

The property $\forall S.\ (S, [\![S]\!]) \in \mathcal{R}_{[\![\cdot]\!]}$ allows us to conclude from pairs of target terms in $\mathcal{R}_{\mathsf{T}}$ and $\mathcal{R}_{[\![\cdot]\!]}$ on pairs of a source and a target term. This is necessary to prove that the encoding satisfies weak PrOC in the “only if”-case, but also ensures that $\mathcal{R}_{[\![\cdot]\!]}$ is a relation that relates source terms with their literal translations. From Theorem 9 and Theorem 14 we can thus conclude that the encoding $(\!|\cdot|\!)_{\mathsf{CCS_p}}^{\pi_\mathsf{p}}$ from $\mathsf{CCS_p}$ into π_p relates a source term $S \in \mathcal{P}_C$ and its literal translation $(\!|S|\!)_{\mathsf{CCS_p}}^{\pi_\mathsf{p}}$ by a probabilistic correspondence simulation.

As derived in [30], weak operational correspondence does not directly map to a well-known, i.e., standard, kind of simulation relation, but is linked to the new relation correspondence simularity. Correspondence simularity is in between the standard simulation relations coupled similarity (see e.g. [27, 3]) and bisimilarity (see e.g. [23]).

Coupled similarity is strictly weaker than bisimilarity. As pointed out in [27], in contrast to bisimilarity it allows for intermediate states in simulations: states that cannot be identified with states of the simulated term. Each symmetric coupled simulation is a bisimulation.

▶ **Definition 15** (Coupled Simulation). *A relation $\mathcal{R}$ is a* (weak reduction) coupled simulation *if both $(\exists Q'.\ Q \Longmapsto Q' \wedge (P', Q') \in \mathcal{R})$ and $(\exists Q'.\ Q \Longmapsto Q' \wedge (Q', P') \in \mathcal{R})$ whenever $(P, Q) \in \mathcal{R}$ and $P \Longmapsto P'$. Two terms are* coupled similar *if they are related by a coupled simulation in both directions.*

Just as coupled similarity, correspondence similarity allows for intermediate states that result e.g. from partial commitments, but in contrast to coupled similarity these intermediate states are not necessarily covered in the relation. Correspondence similarity is obviously strictly weaker than bisimilarity, but as shown in [30] it implies coupled similarity. The same holds for the probabilistic variants of correspondence similarity and coupled similarity, where probabilistic coupled similarity is the adaptation of coupled similarity to distributions using Definition 2.

▶ **Definition 16** (Probabilistic Coupled Simulation). *A relation $\mathcal{R}$ is a* (weak) probabilistic (reduction) coupled simulation *if we have both $\left(\exists \Theta.\ Q \Longmapsto \Theta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}\right)$ and $\left(\exists \Theta.\ Q \Longmapsto \Theta \wedge (\Theta, \Delta) \in \overline{\mathcal{R}}\right)$ whenever $(P, Q) \in \mathcal{R}$ and $P \Longmapsto \Delta$. Two terms are* probabilistic coupled similar *if they are related by a probabilistic coupled simulation in both directions.*

For each probabilistic correspondence simulation $\mathcal{R}$ there exists a probabilistic coupled simulation $\mathcal{R}'$ such that $\forall (P, Q) \in \mathcal{R}.\ (P, Q), (Q, P) \in \mathcal{R}'$.

Intuitively, coupled similarity (and also probabilistic coupled similarity) is the strictest standard simulation relation that allows for intermediate states. Accordingly, that a weak probabilistically operationally corresponding encoding ensures that a source term is probabilistic coupled similar to its literal translation is indeed an interesting property (see e.g. [27, 3] for the relevance of coupled similarity). This proves that our version of weak PrOC is meaningful. As described in [30], the combination with the criteria divergence reflection and success sensitiveness further strengthens the induced relation between source and target, i.e., we obtain a divergence reflecting, success sensitive, probabilistic correspondence (or coupled) simulation to relate source terms and their literal translations.

Weak operational correspondence is very flexible and allows us to encode source term concepts that have no direct counterpart in the target. As discussed in [27, 17, 29], relating source terms and their literal translations by a bisimulation does not allow for intermediate states, i.e., for bisimulation we need stricter encodability criteria as discussed next.

6 (Strong) Probabilistic Operational Correspondence

In this section we proceed in the opposite direction. We start with a standard simulation relation and derive a version of PrOC that induces such a relation between source terms and their translations. Often bisimilarity (see e.g. [23]) is considered as *the* standard relation between processes. Accordingly, we start from a probabilistic version of bisimilarity, namely probabilistic barbed bisimilarity as introduced and analysed in [7].

▶ **Definition 17** (Probabilistic Barbed Bisimulation, [7]). *An equivalence $\mathcal{R}$ is a* probabilistic barbed bisimulation *if for each $(P, Q) \in \mathcal{R}$:*

- $P \longmapsto \Delta$ *implies* $\exists \Theta.\ Q \Longmapsto \Theta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$
- *for each atomic action a, if $P{\downarrow_a}$ then $Q{\Downarrow_a}$*

Two terms are probabilistic barbed bisimilar *if a probabilistic bisimulation relates them.*

Here $P{\downarrow_a}$ if $P \xrightarrow{a} \Delta$, and $\Delta{\downarrow_a}$ if $P{\downarrow_a}$ for all $P \in \lceil \Delta \rceil$ with $a \in \mathcal{N} \cup \overline{\mathcal{N}}$. Moreover, $P{\Downarrow_a} = \exists \Delta.\ P \Longmapsto \Delta \wedge \Delta{\downarrow_a}$ and $\Delta{\Downarrow_a} = \exists \Delta'.\ \Delta \Longmapsto \Delta' \wedge \Delta'{\downarrow_a}$. We use this notion of barbs on our source language $\mathsf{CCS_p}$. Accordingly, barbs $\cdot{\downarrow_a}$ are defined via labelled steps. The versions of operational correspondence that we considered so far do not use labelled but only reduction steps. This is because the treatment of labels in encodings can be difficult. For instance the labels in Mobile Ambients ([5]) are technically and conceptionally very different from the labels in the Join-calculus ([11]) and both kinds of labels are technically and conceptionally very different from labels in CCS or the π-calculus. The consideration of labelled steps is only meaningful if the considered source and target language use similar kinds of labels. Because of that, operational correspondence usually considers reduction steps only. If the languages have similar notions of barbs, success sensitiveness can be replaced by barbed sensitiveness to establish this connection. We remove the condition on barbs.

▶ **Definition 18** (Probabilistic Bisimulation). *A relation $\mathcal{R}$ is a* probabilistic (reduction) bisimulation *if for each $(P, Q) \in \mathcal{R}$:*

- $P \Longmapsto \Delta$ *implies* $\exists \Theta.\ Q \Longmapsto \Theta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$
- $Q \Longmapsto \Theta$ *implies* $\exists \Delta.\ P \Longmapsto \Delta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$

Two terms are probabilistic bisimilar *if a probabilistic bisimulation relates them.*

In comparison to Definition 17 we also replace $P \longmapsto \Delta$ by $P \Longmapsto \Delta$, remove the condition of $\mathcal{R}$ being an equivalence, and add the symmetric case of the first condition. These adaptations have little consequence on the derived relation, but provide a structure more closely related to operational correspondence.

Note that the above notion of probabilistic bisimulation without barbs is trivial. However, also operational correspondence is a trivial encodability relation unless we combine it with success sensitiveness. We discuss the combination with success or barbs below.

We want to derive a version of PrOC for probabilistic bisimilarity. As we learnt in Section 5, the first condition

$$P \Longmapsto \Delta \longrightarrow \exists \Theta.\, Q \Longmapsto \Theta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$$

for the left part of pairs has to be linked with completeness. Since it is identical to the first condition of probabilistic correspondence simulation in Definition 12 we can link it to the same version of completeness:

$$\forall S, \Delta_S.\, S \Longmapsto \Delta_S \longrightarrow \left(\exists \Delta_T.\, [\![S]\!] \Longmapsto \Delta_T \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\right)$$

The second condition

$$Q \Longmapsto \Theta \longrightarrow \exists \Delta.\, P \Longmapsto \Delta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$$

of Definition 18 is simpler than the second condition of probabilistic correspondence simulation in Definition 12. The part that allows for intermediate states is missing. Thus, we link it to a similarly simplified version of soundness:

$$\forall S, \Delta_T.\, [\![S]\!] \Longmapsto \Delta_T \longrightarrow \left(\exists \Delta_S.\, S \Longmapsto \Delta_S \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\right)$$

We denote the result as probabilistic operational correspondence or shortly as PrOC.

▶ **Definition 19** (Probabilistic Operational Correspondence). *An encoding* $[\![\cdot]\!] : \mathcal{P}_{\mathsf{S}} \to \mathcal{P}_{\mathsf{T}}$ *is* probabilistic operationally corresponding *(PrOC) w.r.t.* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *if it is:*

Probabilistic Complete:

$\forall S, \Delta_S.\, S \Longmapsto \Delta_S$ *implies* $\left(\exists \Delta_T.\, [\![S]\!] \Longmapsto \Delta_T \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\right)$

Probabilistic Sound:

$\forall S, \Delta_T.\, [\![S]\!] \Longmapsto \Delta_T$ *implies* $\left(\exists \Delta_S.\, S \Longmapsto \Delta_S \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\right)$

Of course we have to formally establish the described connection between probabilistic bisimilarity and PrOC.

▶ **Theorem 20** (PrOC). $[\![\cdot]\!]$ *is probabilistically operationally corresponding w.r.t. a preorder* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *that is a probabilistic bisimulation iff* $\exists \mathcal{R}_{[\![\cdot]\!]}.\, \left(\forall S.\, (S, [\![S]\!]) \in \mathcal{R}_{[\![\cdot]\!]}\right) \wedge \mathcal{R}_{\mathsf{T}} = \mathcal{R}_{[\![\cdot]\!]}\restriction_{\mathcal{P}_{\mathsf{T}}} \wedge \left(\forall S, T.\, (S, T) \in \mathcal{R}_{[\![\cdot]\!]} \longrightarrow ([\![S]\!], T) \in \mathcal{R}_{\mathsf{T}}\right) \wedge \mathcal{R}_{[\![\cdot]\!]}$ *is a preorder and a probabilistic bisimulation.*

The proof of Theorem 20 is as challenging as the proof of Theorem 14 but fortunately we can reuse the main proof strategy.

In the same way, we can also link strong probabilistic bisimilarity and a strong version of PrOC. We obtain strong probabilistic bisimilarity by considering only single steps.

▶ **Definition 21** (Strong Probabilistic Bisimulation). *A relation* $\mathcal{R}$ *is a* strong probabilistic (reduction) bisimulation *if for each* $(P, Q) \in \mathcal{R}$*:*

- $P \longmapsto \Delta$ *implies* $\exists \Theta.\, Q \longmapsto \Theta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$
- $Q \longmapsto \Theta$ *implies* $\exists \Delta.\, P \longmapsto \Delta \wedge (\Delta, \Theta) \in \overline{\mathcal{R}}$

Two terms are strong probabilistic bisimilar *if a strong probabilistic bisimulation relates them.*

Strong probabilistic operational correspondence is obtained in a similar way from PrOC by considering only single steps.

▶ **Definition 22** (Strong Probabilistic Operational Correspondence). *An encoding* $[\![\cdot]\!] : \mathcal{P}_{\mathsf{S}} \to \mathcal{P}_{\mathsf{T}}$ *is* strongly probabilistic operationally corresponding *(strong PrOC) w.r.t.* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *if it is:*

Strongly Probabilistic Complete:

$$\forall S, \Delta_S.\ S \longmapsto \Delta_S \text{ implies } \big(\exists \Delta_T.\ [\![S]\!] \longmapsto \Delta_T \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\big)$$

Strongly Probabilistic Sound:

$$\forall S, \Delta_T.\ [\![S]\!] \longmapsto \Delta_T \text{ implies } \big(\exists \Delta_S.\ S \longmapsto \Delta_S \wedge ([\![\Delta_S]\!], \Delta_T) \in \overline{\mathcal{R}_{\mathsf{T}}}\big)$$

Finally, we establish the connection between strong PrOC and strong probabilistic bisimilarity in Theorem 23. The proofs of Theorem 20 and Theorem 23 can be found in [33].

▶ **Theorem 23** (Strong PrOC). $[\![\cdot]\!]$ *is strongly probabilistically operationally corresponding w.r.t. a preorder* $\mathcal{R}_{\mathsf{T}} \subseteq \mathcal{P}_{\mathsf{T}}^2$ *that is a strong probabilistic bisimulation iff* $\exists \mathcal{R}_{[\![\cdot]\!]}.\ \big(\forall S.\ (S, [\![S]\!]) \in \mathcal{R}_{[\![\cdot]\!]}\big) \wedge \mathcal{R}_{\mathsf{T}} = \mathcal{R}_{[\![\cdot]\!]}{\restriction_{\mathcal{P}_{\mathsf{T}}}} \wedge \big(\forall S, T.\ (S, T) \in \mathcal{R}_{[\![\cdot]\!]} \longrightarrow ([\![S]\!], T) \in \mathcal{R}_{\mathsf{T}}\big) \wedge \mathcal{R}_{[\![\cdot]\!]}$ *is a preorder and a strong probabilistic bisimulation.*

If (strong) bisimilarity is *the* standard reference relation, i.e., if we usually do not record differences between terms that cannot be observed by (strong) probabilistic bisimilarity, then an encoding that ensures that source terms and their translations are (strongly) probabilistic bisimilar strongly validates the claim that the target language is at least as expressive as the source language. In this sense PrOC and strong PrOC are strict but also very meaningful criteria.

Again the combination with the criteria divergence reflection and success sensitiveness further strengthens the induced relation between source and target (see [30]), i.e., we obtain a divergence reflecting, success sensitive, (strong) probabilistic bisimulation to relate source terms and their literal translations.

As discussed above, the consideration of labels and barbs is difficult in the context of encodings unless the source and target language have very similar notions of labels and barbs. In our source language $\mathsf{CCS_p}$, labels are of the form x or $\overline{x}$, whereas by [38] our target language π_{p} uses labels of the form $x\mathtt{in}_i\langle\tilde{y}_i\rangle$, $\overline{x}\mathtt{in}_i\langle\tilde{y}_i\rangle$, $x\langle\tilde{y}\rangle$, and $\overline{x}\langle\tilde{y}\rangle$. The main difference are the transmitted values and variables for reception. However, for all terms that are created by our encoding function $\llparenthesis\cdot\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ in Definition 6, i.e., all terms in Δ_T such that $\llparenthesis S\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}} \Longmapsto \Delta_T$ for some source term S, all visible labels are of the form $x\mathtt{in}_i\langle\rangle$ or $\overline{x}\mathtt{in}_i\langle\rangle$. Labels of the form $x\langle\tilde{y}\rangle$ and $\overline{x}\langle\tilde{y}\rangle$ are used to emulate the unfolding of recursion but the respective channel names C_i are restricted, i.e., not visible. Moreover, CCS-like barbs, i.e., barbs without values or variables, are often also used for variants of the π-calculus.

This observation allows us to define a suitable notion of barbs for our target language π_{p}[1]. Let $P{\downarrow_x}$ if $P\left\{\xrightarrow[p_i]{x\mathtt{in}_i\langle\rangle} T_i\right\}_{i\in I}$, $P{\downarrow_{\overline{x}}}$ if $P\left\{\xrightarrow[p_i]{\overline{x}\mathtt{in}_i\langle\rangle} T_i\right\}_{i\in I}$, and $\Delta{\downarrow_a}$ if $P{\downarrow_a}$ for all $P \in \lceil\Delta\rceil$ with $a \in \mathcal{N} \cup \overline{\mathcal{N}}$. Moreover, $P{\Downarrow_a} = \exists \Delta.\ P \Longmapsto \Delta \wedge \Delta{\downarrow_a}$ and $\Delta{\Downarrow_a} = \exists \Delta'.\ \Delta \Longmapsto \Delta' \wedge \Delta'{\downarrow_a}$.

Note that our encoding $\llparenthesis\cdot\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ does not translate an input on x to a step with label $x\mathtt{in}_i\langle\rangle$ but with label $\varphi_{\llparenthesis\cdot\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}}(x)\mathtt{in}_i\langle\rangle$, i.e., we should not forget the renaming policy. Accordingly we compare the reachability of a barb n in the source, i.e., $S{\Downarrow_n}$, with the reachability of the translated barb $\varphi_{\llparenthesis\cdot\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}}(n)$ in the translation, i.e., $\llparenthesis S\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}{\Downarrow_{\varphi_{\llparenthesis\cdot\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}}(n)}}$. With these notions of barbs on the source and target language, we can show that our encoding $\llparenthesis\cdot\rrparenthesis^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ is barb sensitive.

[1] Remember that the semantics of π_{p} is given as a Segala automaton, i.e., $P\left\{\xrightarrow[p_i]{\alpha_i} T_i\right\}_{i\in I}$ means that for every $i \in I$ the term P can do a step to T_i with label α_i and probability p_i (see [38, 33]).

▶ **Lemma 24** (Barb Sensitiveness, $(\!|\cdot|\!)^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}/[\![\cdot]\!]^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$). *For every S and all $n \in \mathcal{N} \cup \overline{\mathcal{N}}$, $S{\Downarrow_n}$ iff $(\!|S|\!)^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}{\Downarrow_{\varphi_{(\!|\cdot|\!)^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}}(n)}}$.*

If we combine, as described in [30], barb sensitiveness with PrOC, we obtain that all source terms S are related to $[\![S]\!]$ by a probabilistic barbed bisimulation as described in Definition 17. However, since our encoding $(\!|\cdot|\!)^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ satisfies only weak PrOC and not PrOC, S and $(\!|S|\!)^{\pi_{\mathsf{p}}}_{\mathsf{CCS_p}}$ are related by a probabilistic barbed correspondence (or coupled) simulation.

7 Conclusions

We provided three notions of probabilistic operational correspondence:

1. Weak Probabilistic Operational Correspondence (weak PrOC), where single source term steps can be translated by sequences of target term steps and intermediate states in the translation are possible.
2. Probabilistic Operational Correspondence (PrOC), where single source term steps can be translated by sequences of target term steps but intermediate states are forbidden.
3. Strong Probabilistic Operational Correspondence (strong PrOC), where a single source term step has to be translated to a single target term step.

We proved that strong PrOC induces a strong probabilistic bisimulation between source terms and their literal translations, i.e., strong PrOC is a very strict criterion that ensures a close connection between the source and target language. In contrast weak PrOC induces a probabilistic correspondence (or coupled) simulation between source terms and their literal translations. This allows for pre- and post-processing steps in the encoding and even for intermediate states, i.e., for more flexibility in the creation of encoding functions.

Related Work. There are several papers such as e.g. [4, 24, 14, 15, 13, 12, 36, 37, 16, 25, 26, 29, 30] that study quality criteria for encodings in the traditional setting. As far as we know, there are no studies of quality criteria for encodings between probabilistic systems.

Probabilistic versions of bisimulation for process calculi are studied e.g. in [18, 20, 34, 1, 2, 7]. Encodings between concrete probabilistic process calculi are studied e.g. in [35, 19]. They argue for the quality of the specific presented encodings but do not derive quality criteria for encodings in general. For instance [35] compares two versions of the probabilistic π-calculus (one with mixed choice and one with only separate choice). Since the two versions of the considered language are close, they prove the quality of their encoding by showing a direct correspondence between labelled steps of the respective source and target language. Essentially they prove a labelled variant of weak PrOC. However, as discussed above the consideration of labels in encodings is difficult, because different languages usually have very different notions of labels. Because of that, we use reduction steps and barbs or success for our general formulation of quality criteria. Moreover, since [35, 19] do not present general quality criteria for encodings, they also do not discuss the quality of such criteria.

We are focusing on encodings between process calculi. Instead [38] connects the probabilistic π-calculus and event structures. Therefore, they show an operational correspondence between the semantics of the π-calculus and event structures (see Theorem 6.3 in Section 6.2 of [38]). Their formulation of operational correspondence is basically a variant of strong PrOC in Definition 22.

Further Work. Our original motivation to study versions of PrOC steamed from quantum based systems. As probabilistic versions of simulation relations are essential for studying quantum based systems (see e.g. [10, 9, 8, 6]), this work supports the development of formal methods for quantum based systems.

References

1 C. Baier and H. Hermanns. Weak Bisimulation for Fully Probabilistic Processes. In *Computer Aided Verification*, pages 119–130. Springer, 1997. doi:10.1007/3-540-63166-6_14.

2 C. Baier, H. Hermanns, J.-P. Katoen, and V. Wolf. Bisimulation and Simulation Relationsfor Markov Chains. *Electronic Notes in Theoretical Computer Science*, 162:73–78, 2006. doi:10.1016/j.entcs.2005.12.078.

3 B. Bisping, U. Nestmann, and K. Peters. Coupled Similarity: the first 32 years. *Acta Informatica*, 57:439–463, 2019. doi:10.1007/s00236-019-00356-4.

4 G. Boudol. Asynchrony and the π-calculus (note). Note, INRIA, 1992.

5 L. Cardelli and A.D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. doi:10.1016/S0304-3975(99)00231-5.

6 Y. Deng. Bisimulations for Probabilistic and Quantum Processes. In *Proc. of CONCUR*, volume 118, pages 2:1–2:14, 2018. doi:10.4230/LIPIcs.CONCUR.2018.2.

7 Y. Deng and W. Du. Probabilistic Barbed Congruence. *Electronic Notes in Theoretical Computer Science*, 190(3):185–203, 2007. doi:10.1016/j.entcs.2007.07.011.

8 Y. Deng and Y. Feng. Open Bisimulation for Quantum Processes. In *Proc. of TCS*, volume 7604, pages 119–133. Springer, 2012. doi:10.1007/978-3-642-33475-7_9.

9 Y Feng, Y. Deng, and M. Ying. Symbolic Bisimulation for Quantum Processes. *CoRR*, 2012. doi:10.48550/arXiv.1202.3484.

10 Y. Feng, R. Duan, and M. Ying. Bisimulation for Quantum Processes. *ACM Trans. Program. Lang. Syst.*, 34(4):17:1–17:43, 2012. doi:10.1145/2400676.2400680.

11 C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proc. of POPL*, pages 372–385. ACM, 1996. doi:10.1145/237721.237805.

12 Y. Fu. Theory of Interaction. *Theoretical Computer Science*, 611:1–49, 2016. doi:10.1016/j.tcs.2015.07.043.

13 Y. Fu and H. Lu. On the expressiveness of interaction. *Theoretical Computer Science*, 411(11-13):1387–1451, 2010. doi:10.1016/j.tcs.2009.11.011.

14 D. Gorla. Towards a Unified Approach to Encodability and Separation Results for Process Calculi. In *Proc. of CONCUR*, volume 5201, pages 492–507. Springer, 2008. doi:10.1007/978-3-540-85361-9_38.

15 D. Gorla. Towards a Unified Approach to Encodability and Separation Results for Process Calculi. *Information and Computation*, 208(9):1031–1053, 2010. doi:10.1016/j.ic.2010.05.002.

16 D. Gorla and U. Nestmann. Full abstraction for expressiveness: history, myths and facts. *Mathematical Structures in Computer Science*, pages 1–16, 2014. doi:10.1017/S0960129514000279.

17 M. Hatzel, C. Wagner, K. Peters, and U. Nestmann. Encoding CSP into CCS. In *Proc. of EXPRESS/SOS*, volume 7, pages 61–75, 2015. doi:10.4204/EPTCS.190.5.

18 C. Jou and S.A. Smolka. Equivalences, Congruences, and Complete Axiomatizations for Probabilistic Processes. In *Proc. of CONCUR*, pages 367–383, 1990. doi:10.1007/BFb0039071.

19 M. Kwiatkowska, G. Norman, D. Parker, and M.G. Vigliotti. Probabilistic Mobile Ambients. *Theoretical Computer Science*, 410:1272–1303, 2009. doi:10.1016/j.tcs.2008.12.058.

20 K.G. Larsen and A. Skou. Bisimulation through Probabilistic Testing. *Information and Computation*, 94(1):1–28, 1991. doi:10.1016/0890-5401(91)90030-6.

21 R. Milner. *Communication and Concurrency.* Prentice-Hall, Inc., 1989.

22 R. Milner. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999.

23 R. Milner and D. Sangiorgi. Barbed Bisimulation. In *Automata, Languages and Programming*, pages 685–695, 1992.

24 C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. doi:10.1017/S0960129503004043.

25 J. Parrow. Expressiveness of Process Algebras. *Electronic Notes in Theoretical Computer Science*, 209:173–186, 2008. doi:10.1016/j.entcs.2008.04.011.

26 J. Parrow. General conditions for full abstraction. *Mathematical Structures in Computer Science*, 26(4):655–657, 2014. doi:10.1017/S0960129514000280.

27 J. Parrow and P. Sjödin. Multiway Synchronization Verified with Coupled Simulation. In *Proc. of CONCUR*, volume 630 of *LNCS*, pages 518–533, 1992. doi:10.1007/BFb0084813.

28 K. Peters. *Translational Expressiveness*. PhD thesis, TU Berlin, 2012. URL: http://opus.kobv.de/tuberlin/volltexte/2012/3749/.

29 K. Peters. Comparing Process Calculi Using Encodings. In *Proc. of EXPRESS/SOS*, EPTCS, pages 19–38, 2019. doi:10.48550/arXiv.1908.08633.

30 K. Peters and R. van Glabbeek. Analysing and Comparing Encodability Criteria. In *Proc. of EXPRESS/SOS*, volume 190 of *EPTCS*, pages 46–60, 2015. doi:10.4204/EPTCS.190.4.

31 G.D. Plotkin. A structural approach to operational semantics. *Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

32 D. Sangiorgi. π-Calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(1):235–274, 1996. doi:10.1016/0304-3975(96)00075-8.

33 Anna Schmitt and Kirstin Peters. Probabilistic Operational Correspondence (Technical Report). Technical report, TU Darmstadt and Augsburg University, 2023. doi:10.48550/arXiv.2307.05218.

34 R. Segala and N. Lynch. Probabilistic Simulations for Probabilistic Processes. In *Proc. of CONCUR*, pages 481–496, 1994. doi:10.1007/978-3-540-48654-1_35.

35 P. Sylvain and C. Palamidessi. Expressiveness of Probabilistic π-calculus. *Electronic Notes in Theoretical Computer Science*, 164:119–136, 2006. doi:10.1016/j.entcs.2006.07.015.

36 R. van Glabbeek. Musings on Encodings and Expressiveness. In *Proc. of EXPRESS/SOS*, volume 89, pages 81–98, 2012. doi:10.4204/EPTCS.89.7.

37 R. van Glabbeek. A Theory of Encodings and Expressiveness (Extended Abstract). In *Proc. of FOSSACS*, volume 10803, pages 183–202. Springer, 2018. doi:10.1007/978-3-319-89366-2_10.

38 D. Varacca and N. Yoshida. Probabilistic pi-Calculus and Event Structures. *Electronic Notes in Theoretical Computer Science*, 190(3):147–166, 2007. doi:10.1016/j.entcs.2007.07.009.

A Game of Pawns

Guy Avni
University of Haifa, Israel

Pranav Ghorpade
Chennai Mathematical Institute, India

Shibashis Guha
Tata Institute of Fundamental Research, Mumbai, India

Abstract

We introduce and study *pawn games*, a class of two-player zero-sum turn-based graph games. A turn-based graph game proceeds by placing a token on an initial vertex, and whoever *controls* the vertex on which the token is located, chooses its next location. This leads to a path in the graph, which determines the winner. Traditionally, the control of vertices is predetermined and fixed. The novelty of pawn games is that control of vertices changes dynamically throughout the game as follows. Each vertex of a pawn game is *owned* by a *pawn*. In each turn, the pawns are partitioned between the two players, and the player who *controls* the pawn that owns the vertex on which the token is located, chooses the next location of the token. Control of pawns changes dynamically throughout the game according to a fixed mechanism. Specifically, we define several *grabbing*-based mechanisms in which control of at most one pawn transfers at the end of each turn. We study the complexity of solving pawn games, where we focus on reachability objectives and parameterize the problem by the mechanism that is being used and by restrictions on pawn ownership of vertices. On the positive side, even though pawn games are exponentially-succinct turn-based games, we identify several natural classes that can be solved in PTIME. On the negative side, we identify several EXPTIME-complete classes, where our hardness proofs are based on a new class of games called Lock & Key games, which may be of independent interest.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Graph games, Reachability games, Pawn games, Dynamic vertex control

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.16

Related Version *Full Version*: `https://arxiv.org/pdf/2305.04096.pdf` [7]

Funding *Guy Avni*: Partially supported by ISF grant No. 1679/21.
Shibashis Guha: Partially supported by the Indian Science and Engineering Research Board (SERB) grant SRG/2021/000466.

1 Introduction

Two-player zero-sum *graph games* constitute a fundamental class of games [5] with applications, e.g., in *reactive synthesis* [26], multi-agent systems [4], and more. A graph game is played on a directed graph $\langle V, E\rangle$, where $V = V_1 \cup V_2$ is a fixed partition of the vertices. The game proceeds as follows. A token is initially placed on some vertex. When the token is placed on $v \in V_i$, for $i \in \{1, 2\}$, Player i chooses u with $\langle v, u\rangle \in E$ to move the token to. The outcome of the game is an infinite path, called a *play*. We focus on *reachability* games: Player 1 wins a play iff it visits a set of target vertices $T \subseteq V$.

In this paper, we introduce *pawn games*, which are graph games in which the control of vertices changes dynamically throughout the game as follows. The arena consists of d *pawns*. For $1 \leq j \leq d$, Pawn j *owns* a set of vertices V_j. Throughout the game, the pawns are distributed between the two players, and in each turn, the control of pawns determines which player moves the token. Pawn control may be updated after moving the token by running a

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 16; pp. 16:1–16:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

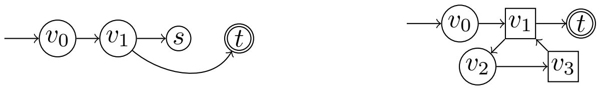


Figure 1 Left: The pawn game $\mathcal{G}_1$; a non-monotonic game under optional-grabbing. Right: The pawn game $\mathcal{G}_2$ in which Player 1 wins from $\langle v_0, \{v_0, v_1\}\rangle$, but must visit v_1 twice.

predetermined *mechanism*. Formally, a *configuration* of a pawn game is a pair $\langle v, P\rangle$, where v denotes the position of the token and P the set of pawns that Player 1 controls. The player who moves the token is determined according to P: if Player 1 controls a pawn that owns v, then Player 1 moves. Specifically, when each vertex is owned by a unique pawn, i.e., $V_1, \ldots, V_d$ partitions V, then Player 1 moves iff he controls the pawn that owns v. We consider the following mechanisms for exchanging control of pawns. For $i \in \{1, 2\}$, we denote by $-i = 3 - i$ the "other player".

Optional grabbing. For $i \in \{1, 2\}$, following a Player i move, Player $-i$ has the *option* to *grab* one of Player i's pawns; namely, transfer one of the pawns that Player $-i$ to his control.

Always grabbing. For $i \in \{1, 2\}$, following *every* Player i move, Player $-i$ grabs one of Player i's pawns.

Always grabbing or giving. Following a Player i move, Player $-i$ either grabs one of Player i's pawns or gives her one of his pawns.

k-grabbing. For $k \in \mathbb{N}$, Player 1 can grab at most k pawns from Player 2 throughout the game. In each round, after moving the token, Player 1 has the option of grabbing one of the pawns that is controlled by Player 2. A grabbed pawn stays in the control of Player 1 for the remainder of the game. Note the asymmetry: only Player 1 grabs pawns.

Note that players in pawn games have two types of actions: moving the token and transferring control of pawns. We illustrate the model and some interesting properties of it.

▶ **Example 1.** Consider the game $\mathcal{G}_1$ in Fig. 1(left). We consider optional-grabbing and the same reasoning applies for always-grabbing. Each vertex is owned by a unique pawn, and Player 1's target is t. Note that Player 2 wins if the game reaches s. We claim that $\mathcal{G}_1$ is *non-monotonic*: increasing the set of pawns that Player 1 initially controls is "harmful" for him. Formally, Player 1 wins from configuration $\langle v_0, \emptyset\rangle$, i.e., when he initially does not control any pawns, but loses from $\langle v_0, \{v_0\}\rangle$, i.e., when controlling v_0. Indeed, from $\langle v_0, \emptyset\rangle$, Player 2 initially moves the token from v_0 to v_1, Player 1 then uses his option to grab v_1, and wins by proceeding to t. Second, from $\langle v_0, \{v_0\}\rangle$, Player 1 makes the first move and thus cannot grab v_1. Since Player 2 controls v_1, she wins by proceeding to s. In Thm. 5 and 18, we generalize this observation and show, somewhat surprisingly, that if a player wins from the current vertex v, then he wins from v with fewer pawns as long as if he controlled v previously, then he maintains control of v.

Consider the game $\mathcal{G}_2$ in Fig. 1 (right). We consider optional-grabbing, each vertex is owned by a unique pawn, and Player 1's target is t. We claim that Player 1 wins from configuration $\langle v_0, \{v_0, v_2\}\rangle$ and Player 2 can force the game to visit v_1 twice. This differs from turn-based games in which if Player 1 wins, he can force winning while visiting each vertex at most once. To illustrate, consider the following outcome. Player 1 makes the first move, so he cannot grab v_1. Player 2 avoids losing by moving to v_2. Player 1 will not grab, move to v_3, Player 2 moves to v_1, then Player 1 grabs v_1 and proceeds to t. We point out that no loop is closed in the explicit *configuration graph* that corresponds to $\mathcal{G}_2$.

Applications

Pawn games model multi-agent settings in which the agent who acts in each turn is not predetermined. We argue that such settings arise naturally.

Quantitative shield synthesis. It is common practice to model an *environment* as a Kripke structure (e.g. [27]), which for sake of simplicity, we will think of as a graph in which vertices model environment states and edges model actions. A *policy* chooses an outgoing edge from each vertex. A popular technique to obtain policies is *reinforcement learning* (RL) [29] whose main drawback is lack of worst-case guarantees [13]. In order to regain safety at runtime, a *shield* [19, 6, 13] is placed as a proxy: in each point in time, it can alter the action of a policy. The goal in *shield synthesis* is to synthesize a shield *offline* that ensures safety at runtime while minimizing interventions. We suggest a procedure to synthesize shields based on k-grabbing pawn games. Player 2 models an unknown policy. We set his goal to reaching an unsafe state. Player 1 (the shield) ensures safety by grabbing at most k times. Grabbing is associated with a shield intervention. Note that once the shield intervenes in a vertex v, it will choose the action at v in subsequent turns. An optimal shield is obtained by finding the minimal k for which Player 1 has a winning strategy.

We describe other examples that can be captured by a k-grabbing pawn game in which Player 1 models an “authority” that has the “upper hand”, and aims to maximize freedom of action for Player 2 while using grabs to ensure safety. Consider a concurrent system in which Player 2 models a scheduler and Player 1 can force synchronization, e.g., by means of “locks” or “fences” in order to maintain correctness (see [14]). Synchronization is minimized in order to maximize parallelism and speed. As another example, Player 1 might model an operating system that allows freedom to an application and blocks only unsafe actions. As a final example, in [2], synthesis for a safety specification was enriched with “advice” given by an external policy for optimizing a soft quantitative objective. Again, the challenge is how to maximize accepting advice while maintaining safety.

Modelling crashes. A *sabotage game* [31] is a two-player game which is played on a graph. Player 1 (the Runner) moves a token throughout the graph with the goal of reaching a target set. In each round, Player 2 (the Saboteur) crashes an edge from the graph with the goal of preventing Player 1 from reaching his target. Crashes are a simple type of fault that restrict Player 1’s actions. A malicious fault (called *byzantine faults* [21]) actively tries to harm the network, e.g., by moving away from the target. Pawn games can model sabotage games with byzantine faults: each vertex (router) is owned by a unique pawn, all pawns are initially owned by Player 1, and a Player 2 grab corresponds to a byzantine fault. Several grabbing mechanisms are appealing in this context: k-grabbing restricts the number of faults and optional- and always-grabbing accommodate repairs of routers.

Our results

We distinguish between three types of ownership of vertices. Let $V = V_1 \cup \ldots \cup V_d$ be a set of vertices, where for $j \in \{1, \ldots, d\}$, Pawn j owns the vertices in V_j. In *one vertex per pawn* (OVPP) games, each pawn owns exactly one vertex, thus V_j is a singleton, for all $j \in \{1, \ldots, d\}$. In *multiple vertices per pawn* (MVPP) games, $V_1, \ldots, V_d$ consists of a partition of V, where the sets might contain more than one vertex. In *overlapping multiple vertices per pawn* (OMVPP) games, the sets might overlap. For example, in the shield synthesis application above, the type of ownership translates to dependencies between interventions:

OVPP models no dependencies, MVPP models cases in which interventions come in "batches", e.g., grabbing control in all states labeled by some predicate, and OMVPP models the case when the batches overlap. We define that Player 1 moves the token from a vertex v iff he controls at least one of the pawns that owns v. Clearly, OMVPP generalizes MVPP, which in turn generalizes OVPP.

We consider the problem of deciding whether Player 1 wins a reachability pawn game from an initial configuration of the game. Our results are summarized below.

Mechanisms	OVPP	MVPP	OMVPP
k-grabbing	PTIME (Thm. 22)	NP-hard (Thm. 23)	PSPACE-C (Thm. 26)
Optional-grabbing	PTIME (Thm. 7)	EXPTIME-C (Thm. 12)	EXPTIME-C (Thm. 12)
Always	PTIME (grab or give; Thm. 21)	PTIME (grab or give; Thm. 21) EXPTIME-C (grab; Thm. 17)	EXPTIME-C (grab; Thm. 17)

Pawn games are succinctly-represented turn-based games. A naive algorithm to solve a pawn game constructs and solves an explicit turn-based game on its configuration graph leading to membership in EXPTIME. We thus find the positive results to be pleasantly surprising; we identify classes of succinctly-represented games that can be solved in PTIME. Each of these algorithms is obtained by a careful and tailored modification to the *attractor-computation* algorithm for turn-based reachability games. For OMVPP k-grabbing, the PSPACE upper bound is obtained by observing that grabs in a winning strategy must be spaced by at most $|V|$ turns, implying that a game ends within polynomial-many rounds (Lem. 25).

Our EXPTIME-hardness proofs are based on a new class of games called *Lock & Key* games and may be of independent interest. A Lock & Key game is a turn-based game that is enriched with a set of locks, where each lock is associated with a key. Each edge is labeled by a subset of locks and keys. A lock can either be *closed* or *open*. An edge that is labeled with a closed lock cannot be crossed. A lock changes state once an edge labeled by its key is traversed. We show two reductions. The first shows that deciding the winner in Lock & Key games is EXPTIME-hardness. Second, we reduce Lock & Key games to MVPP optional-grabbing pawn games. The core of the reduction consists of gadgets that simulate the operation of locks and keys using pawns. Then, we carefully analyze the pawn games that result from applying both reductions one after the other, and show that the guarantees are maintained when using always grabbing instead of optional grabbing. The main difficulty in constructing a winning Player i strategy under always-grabbing from a winning Player i strategy under optional-grabbing is to ensure that throughout the game, both players have *sufficient* and the *correct* pawns to grab (Lem. 16).

Related work

The semantics of pawn games is inspired by the seminal paper [4]. There, the goal is, given a game, an objective O, and a set C of pawns (called "players" there), to decide whether Player 1 (called a "coalition" there) can ensure O when he controls the pawns in C. A key distinction from pawn games is that the set C that is controlled by Player 1 is fixed. The paper introduced a logic called *alternating time temporal logic*, which was later significantly extended and generalized to *strategy logic* [15, 23, 24]. Multi-player games with rational players have been widely studied; e.g., finding Nash equilibrium [30] or subgame perfect equilibrium [12], and rational synthesis [17, 20, 32, 11]. A key distinction from pawn games is that, in pawn games, as the name suggests, the owners of the resources (pawns) have no individual goals and act as pawns in the control of the players. Changes to multi-player graph games in order guarantee existence or improve the quality of an equilibrium have been studied [3, 25, 10]. The key difference from our approach is that there, changes occur *offline*, before the game starts, whereas in pawn games, the transfer of vertex ownership occurs *online*. In *bidding games* [22, 8] (see in particular, *discrete-bidding games* [16, 1, 9]) control of vertices changes online: players have budgets, and in each turn, a *bidding* determines which player moves the token. Bidding games are technically very different from pawn games. While pawn games allow varied and fine-grained mechanisms for transfer of control, bidding games only consider strict auction-based mechanisms, which lead to specialized proof techniques that cannot be applied to pawn games. For example, bidding games are monotonic – more budget cannot harm a player – whereas pawn games are not (see Ex. 1).

2 Preliminaries

For $k \in \mathbb{N}$, we use $[k]$ to denote the set $\{1, \ldots, k\}$. For $i \in \{1, 2\}$, we use $-i = 3 - i$ to refer to the "other player".

Turn-based games

Throughout this paper we consider *reachability* objectives. For general graph games, see for example [5]. A *turn-based game* is $\mathcal{G} = \langle V, E, T \rangle$, where $V = V_1 \cup V_2$ is a set of vertices that is partitioned among the players, $E \subseteq V \times V$ is a set of directed edges, and $T \subseteq V$ is a set of target vertices for Player 1. Player 1's goal is to reach T and Player 2's goal is to avoid T. For $v \in V$, we denote the *neighbors* of v by $N(v) = \{u \in V : E(v, u)\}$. Intuitively, a *strategy* is a recipe for playing a game: in each vertex it prescribes a neighbor to move the token to. Formally, for $i \in \{1, 2\}$, a (memoryless) strategy for Player i is a function $f : V_i \to V$ such that for every $v \in V_i$, we have $f(v) \in N(v)$.[1] An initial vertex $v_0 \in V$ together with two strategies f_1 and f_2 for the players, give rise to a unique *play*, denoted $\pi(v_0, f_1, f_2)$, which is a finite or infinite path in $\mathcal{G}$ and is defined inductively as follows. The first vertex is v_0. For $j \geq 0$, assuming $v_0, \ldots, v_j$ has been defined, then $v_{j+1} = f_i(v_j)$, where $v_j \in V_i$, for $i \in \{1, 2\}$. A Player 1 strategy f_1 is *winning* from $v_0 \in V$ if for every Player 2 strategy f_2, the play $\pi(v_0, f_1, f_2)$ ends in T. Dually, a Player 2 strategy f_2 is winning from $v_0 \in V$ if for every Player 1 strategy f_1, the play $\pi(v_0, f_1, f_2)$ does not visit T.

▶ **Theorem 2** ([18]). *Turn based games are* determined*: from each vertex, one of the players has a (memoryless) winning strategy. Deciding the winner of a game is in PTIME.*

Proof sketch. For completeness, we briefly describe the classic *attractor-computation* algorithm. Consider a game $\langle V, E, T \rangle$. Let $W_0 = T$. For $i \geq 1$, let $W_i = W_{i-1} \cup \{v \in V_1 : N(v) \cap W_i \neq \emptyset\} \cup \{v \in V_2 : N(v) \subseteq W_i\}$. One can prove by induction that W_i consists of the vertices from which Player 1 can force reaching T within i turns. The sequence necessarily reaches a *fixed point* $W^1 = \bigcup_{i \geq 1} W_i$, which can be computed in linear time. Finally, one can show that Player 2 has a winning strategy from each $v \notin W^1$. ◀

Pawn games

A *pawn game* with $d \in \mathbb{N}$ pawns is $\mathcal{P} = \langle V, E, T, \mathcal{M} \rangle$, where $V = V_1 \cup \ldots \cup V_d$ and for $j \in [d]$, V_j denotes the vertices that Pawn j *owns*, E and T are as in turn-based games, and $\mathcal{M}$ is a mechanism for exchanging pawns as we elaborate later. Player 1 wins a play if it reaches T.

[1] We restrict to *memoryless* strategies since these suffice for reachability objectives.

We stress that the set of pawns that he controls when reaching T is irrelevant. We omit $\mathcal{M}$ when it is clear from the context. We distinguish between classes of pawn games based on the type of ownership of vertices:

One Vertex Per Pawn (OVPP). There is a one-to-one correspondence between pawns and vertices; namely, $|V| = d$ and each V_j is singleton, for $j \in [d]$. For $j \in [d]$ and $\{v_j\} = V_j$, we sometimes abuse notation by referring to Pawn j as v_j.

Multiple Vertices Per Pawn (MVPP). Each vertex is owned by a unique pawn but a pawn can own multiple vertices, thus $V_1, \ldots, V_d$ is a partition of V.

Overlapping Multiple Vertices Per Pawn (OMVPP). Each pawn can own multiple vertices and a vertex can be owned by multiple pawns, i.e., we allow $V_i \cap V_j \neq \emptyset$, for $i \neq j$.

Clearly OMVPP generalizes MVPP, which generalizes OVPP. In MVPP, we sometimes abuse notation and refer to a pawn by a vertex that it owns.

A *configuration* of a pawn game is $\langle v, P \rangle$, meaning that the token is placed on a vertex $v \in V$ and $P \subseteq [d]$ is the set of pawns that Player 1 *controls*. Implicitly, Player 2 controls the complement set $\overline{P} = [d] \setminus P$. Player 1 moves the token from $\langle v, P \rangle$ iff he controls at least one pawn that owns v. Note that in OVPP and MVPP, let $j \in [d]$ with $v \in V_j$, then Player 1 moves iff $i \in P$. Once the token moves, we update the control of the pawns by applying $\mathcal{M}$.

From pawn games to turn-based games. We describe the formal semantics of pawn games together with the pawn-exchanging mechanisms by describing the explicit turn-based game that corresponds to a pawn game. For a pawn game $\mathcal{G} = \langle V, E, T, \mathcal{M} \rangle$, we construct the turn-based game $\mathcal{G}' = \langle V', E', T' \rangle$. For $i \in \{1, 2\}$, denote by V'_i Player i's vertices in $\mathcal{G}'$. The vertices of $\mathcal{G}'$ consist of two types of vertices: configuration vertices $C = V \times 2^{[d]}$, and *intermediate* vertices $V \times C$. When $\mathcal{M}$ is k-grabbing, configuration vertices include the remaining number of pawns that Player 1 can grab, as we elaborate below. The target vertices are $T' = \{\langle v, P \rangle : v \in T\}$. We describe E' next. For a configuration vertex $c = \langle v, P \rangle$, we define $c \in V'_1$ iff there exists $j \in P$ such that $v \in V_j$. That is, Player 1 moves from c in $\mathcal{G}'$ iff he moves from c in $\mathcal{G}$. We define the neighbors of c to be the intermediate vertices $\{\langle v', c \rangle : v' \in N(v)\}$. That is, moving the token in $\mathcal{G}'$ from c to $\langle v', c \rangle$ corresponds to moving the token from v to v' in $\mathcal{G}$. Moves from intermediate vertices represent an application of $\mathcal{M}$. We consider the following mechanisms.

Optional grabbing. For $i \in \{1, 2\}$, following a Player i move, Player $-i$ has the option to grab one of Player i's pawns. Formally, for a configuration vertex $c = \langle v, P \rangle \in V'_1$, we have $N(c) \subseteq V'_2$. From $\langle v', c \rangle \in N(c)$, Player 2 has two options: (1) do not grab and proceed to $\langle v', P \rangle$, or (2) grab $j \in P$, and proceed to $\langle v', P \setminus \{j\} \rangle$. The definition for Player 2 is dual.

Always grabbing. For $i \in \{1, 2\}$, following a Player i move, Player $-i$ always has to grab one of Player i's pawns. The formal definition is similar to optional grabbing with the difference that option (1) of not grabbing is not available to the players. We point out that Player $-i$ grabs only after Player i has moved, which in particular implies that Player i controls at least one pawn that Player $-i$ can grab.

Always grabbing or giving. Following a Player i move, Player $-i$ must either grab one of Player i's pawns or *give* her a pawn. The formal definition is similar to always grabbing with the difference that, for an intermediate vertex $\langle v', \langle v, P \rangle \rangle$, there are both neighbors of the form $\langle v', P \setminus \{j\} \rangle$, for $j \in P$, and neighbors of the form $\langle v', P \cup \{j\} \rangle$, for $j \notin P$.

***k*-grabbing.** After each round, Player 1 has the option of grabbing a pawn from Player 2, and at most k grabs are allowed in a play. A configuration vertex in k-grabbing is $c = \langle v, P, r\rangle$, where $r \in [k] \cup \{0\}$ denotes the number of grabs remaining. Intermediate vertices are Player 1 vertices. Let $\langle v', c\rangle \in V'_1$. Player 1 has two options: (1) do not grab and proceed to the configuration vertex $\langle v', P, r\rangle$, or (2) grab $j \notin P$, and proceed to $\langle v', P \cup \{j\}, r - 1\rangle$ when $r > 0$. Note that grabs are not allowed when $r = 0$ and that Pawn j stays at the control of Player 1 for the remainder of the game.

Since pawn games are succinctly-represented turn-based games, Thm. 2 implies determinacy; namely, one of the players wins from each initial configuration. We study the problem of determining the winner of a pawn game, formally defined as follows.

▶ **Definition 3.** *Let $\alpha \in \{OVPP, MVPP, OMVPP\}$ and β be a pawn-grabbing mechanism. The problem α β PAWN-GAMES takes as input an α β pawn game $\mathcal{G}$ and an initial configuration c, and the goal is to decide whether Player 1 wins from c in $\mathcal{G}$.*

A naive algorithm to solve a pawn game applies attractor computation on the explicit turn-based game, which implies the following theorem.

▶ **Theorem 4.** *α β PAWN-GAMES is in EXPTIME, for all values of α and β.*

3 Optional-Grabbing Pawn Games

Before describing our complexity results, we identify a somewhat unexpected property of MVPP optional-grabbing games. Consider a vertex v and two sets of pawns P and P' having $P' \subseteq P$. Intuitively, it is tempting to believe that Player 1 “prefers” configuration $c = \langle v, P\rangle$ over $c' = \langle v, P'\rangle$ since he controls more pawns in c. Somewhat surprisingly, the following theorem shows that the reverse holds (see also Ex. 1). More formally, the theorem states that if Player 1 wins from c, then he also wins from c', under the restriction that if he makes the first move at c (i.e., he controls v in c), then he also makes the first move in c' (i.e., he controls v in c').

▶ **Theorem 5.** *Consider a configuration $\langle v, P\rangle$ of an MVPP optional-grabbing pawn game $\mathcal{G}$. Let $j \in [d]$ such that $v \in V_j$ and $P' \subseteq P$. Assuming that $j \in P$ implies $j \in P'$, if Player 1 wins from $\langle v, P\rangle$, he wins from $\langle v, P'\rangle$. Assuming that $j \in \overline{P'}$ implies $j \in \overline{P}$, if Player 2 wins from $\langle v, P'\rangle$, she wins from $\langle v, P\rangle$.*

Proof. We prove for Player 1 and the proof for Player 2 follows from determinacy. Let $\mathcal{G}$, P, P', $c = \langle v, P\rangle$, and $c' = \langle v, P'\rangle$ be as in the theorem statement. Let $\mathcal{G}'$ be the turn-based game corresponding to $\mathcal{G}$. For $i \geq 0$, let W_i be the set of vertices in $\mathcal{G}'$ from which Player 1 can win in at most i rounds (see Thm. 2). The following claim clearly implies the theorem. Its proof, which proceeds by a careful induction, can be found in the full version.
Claim: Configuration vertices: for $i \geq 0$, if $\langle v, P\rangle \in W_i$, then $\langle v, P'\rangle \in W_i$. Intermediate vertices: for $i \geq 1$ and every vertex $u \in N(v)$, if $\langle u, c\rangle \in W_{i-1}$, then $\langle u, c'\rangle \in W_{i-1}$. ◀

Thm. 5 implies that we can restrict attention to strategies that only “grab locally”. The assumption $j \in P$ implies $j \in P'$ also implies that if Player 1 wins from $\langle v, P\rangle$ when Player 2 controls v then Player 1 also wins from $\langle v, P'\rangle$ since $P' \subseteq P$.

▶ **Corollary 6.** *Consider an MVPP optional-grabbing game. Suppose that Player 1 controls $P \subseteq [d]$, and that Player 2 moves the token to a vertex v owned by Pawn j, i.e., $v \in V_j$. Player 1 has the option to grab. If Player 1 can win by grabbing a pawn $j' \neq j$, i.e., a pawn that does not own the next vertex, he can win by not grabbing at all. Formally, if Player 1 wins from $\langle v, P \cup \{j'\}\rangle$, he also wins from $\langle v, P\rangle$. And dually for Player 2.*

Algorithm 1 Given an OVPP optional-grabbing pawn game $\mathcal{G} = \langle V, E, T\rangle$ and an initial configuration $c = \langle v, P_0\rangle$, determines which player wins $\mathcal{G}$ from c.

```
1:  $W_0 = T$, $i = 0$
2:  while True do
3:      if $v_0 \in W_i$ then return Player 1
4:      $W_{i+1} = W_i \cup \{u : N(u) \subseteq W_i\}$
5:      if $W_i \neq W_{i+1}$ then $i := i + 1$; Continue
6:      $B := \{u : N(u) \cap W_i \neq \emptyset\}$
7:      if $B = \emptyset$ then return Player 2
8:      if $v_0 \in B$ and $v_0 \in P_0$ then return Player 1
9:      $B' := \{u \in B : N(u) \subseteq B \cup W_i\}$
10:     if $B' \neq \emptyset$ then $W_{i+1} := W_i \cup B'$; $i := i + 1$; Continue
11:     $R = \{u : N(u) \subseteq B\}$
12:     if $R \setminus P_0 \neq \emptyset$ then $W_{i+1} = W_i \cup (R \setminus P_0)$; $i := i + 1$
13:     else return Player 2
```

One can show that Thm. 5 and Cor. 6 do not hold for OMVPP optional-grabbing games.

3.1 OVPP: A PTIME algorithm

We turn to study complexity results, and start with the following positive result.

▶ **Theorem 7.** *OVPP optional-grabbing PAWN-GAMES is in PTIME.*

Proof. We describe the intuition of the algorithm, the pseudo-code can be found in Alg. 1, and its correctness is proven in the full version. Recall that in turn-based games (see Thm. 2), the attractor computation iteratively "grows" the set of states from which Player 1 wins: initially $W_0 = T$, and in each iteration, a vertex u is added to W_i if (1) u belongs to Player 2 and all its neighbors belong to W_i or (2) u belongs to Player 1 and it has a neighbor in W_i. In optional-grabbing games, applying attractor computation is intricate since vertex ownership is dynamic. Note that the reasoning behind (1) above holds; namely, if $N(u) \subseteq W_i$, no matter who controls u, necessarily W_i is reached in the next turn. However, the reasoning behind (2) fails. Consider a Player 1 vertex u that has two neighbors $v_1 \in W_i$ and $v_2 \notin W_i$. While u would be in W_{i+1} according to (2), under optional-grabbing, when Player 1 makes the move into u, Player 2 can avoid W_i by grabbing u and proceeding to v_2.

In order to overcome this, our algorithm operates as follows. Vertices that satisfy (1) are added independent of their owner (Line 4). The counterpart of (2) can be seen as two careful steps of attractor computation. First, let B denote the *border* of W_i, namely the vertices who have a neighbor in W_i (Line 6). Second, a vertex u is in W_{i+1} in one of two cases. (i) $u \in B$ and all of its neighbors are in $B \cup W_i$ (Line 10). Indeed, if Player 1 controls u he wins by proceeding to W_i and if Player 2 owns u, she can avoid W_i by moving to B, then Player 1 grabs and proceeds to W_i. (ii) Player 2 controls u in the initial configuration and all of its neighbors are in B (Line 12). Indeed, Player 2 cannot avoid proceeding into B, and following Player 2's move, Player 1 grabs and proceeds to W_i. Finally, note that the algorithm terminates once a fixed point is reached, thus it runs for at most $|V|$ iterations. ◀

3.2 MVPP: EXPTIME-hardness via Lock & Key games

We prove hardness of MVPP optional-grabbing pawn games by reduction through a class of games that we introduce and call *Lock & Key* games, and may be of independent interest. A Lock & Key game is $\mathcal{G} = \langle V, E, T, L, K, \lambda, \kappa \rangle$, where $\langle V, E, T \rangle$ is a turn-based game, $L = \{\ell_1, \ldots, \ell_n\}$ is a set of locks $K = \{k_1, \ldots, k_n\}$ is a set of keys, each ℓ_j is associated to key $k_j \in K$ for $j \in [n]$, and each edge is labeled by a set of locks and keys respectively given by $\lambda : E \to 2^L$ and $\kappa : E \to 2^K$. Note that a lock and a key can appear on multiple edges.

Intuitively, a Lock & Key game is a turn-based game, only that the locks impose restrictions on the edges that a player is allowed to cross. Formally, a configuration of a Lock & Key game is $c = \langle v, A \rangle \in V \times 2^L$, meaning that the token is placed on v and each lock in A is *closed* (all other locks are *open*). When $v \in V_i$, for $i \in \{1, 2\}$, then Player i moves the token as in turn-based games with the restriction that he cannot choose an edge that is labeled by a closed lock, thus $e = \langle v, u \rangle \in E$ is a legal move at c when $\lambda(e) \subseteq (L \setminus A)$. Crossing e updates the configuration of the locks by "turning" all keys that e is labeled with. Formally, let $\langle u, A' \rangle$ be the configuration after crossing e. For $k_j \in \kappa(e)$ ("key k_j is turned"), we have $\ell_j \in A$ iff $\ell_j \notin A'$. For $k_j \notin \kappa(e)$ ("key k_j is unchanged"), we have $\ell_j \in A$ iff $\ell_j \in A'$.

Note that, similar to pawn games, each Lock & Key game corresponds to an exponentially sized two-player turn-based game. Thus, membership in EXPTIME is immediate. For the lower bound, we show a reduction for the problem of deciding whether an *alternating polynomial-space Turing machine* (ATM) accepts a given word.

▶ **Theorem 8.** *Given a Lock & Key game $\mathcal{G}$ and an initial configuration c, deciding whether Player 1 wins from c in $\mathcal{G}$ is* EXPTIME*-complete.*

Proof. We briefly describe the syntax and semantics of ATMs, see for example [28], for more details. An ATM is $\mathcal{A} = \langle Q, \Gamma, \delta, q_0, q_{acc}, q_{rej} \rangle$, where Q is a collection of states that is partitioned into $Q = Q_1 \cup Q_2$ owned by Player 1 and Player 2 respectively, Γ is a tape alphabet, $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}}$ is a transition function, $q_0, q_{acc}, q_{rej} \in Q$ are respectively an initial, accepting, and rejecting states. A configuration of $\mathcal{A}$ is $c = \langle q, i, \langle \gamma_1, \ldots, \gamma_m \rangle \rangle$, meaning that the control state is q, the head position is i, and $\langle \gamma_1, \ldots, \gamma_m \rangle$ is the tape content, where m is polynomial in the length of the input word w. In order to determine whether $\mathcal{A}$ accepts w we construct a (succinctly-represented) turn-based game over the possible configurations of $\mathcal{A}$, the neighbors of a configuration are determined according to δ, and, for $i \in \{1, 2\}$, Player i moves from states in Q_i. We say that $\mathcal{A}$ accepts w iff Player 1 has a winning strategy from the initial configuration for the target state q_{acc}.

Given $\mathcal{A}$ and w, we construct a Lock & Key game $\mathcal{G} = \langle V, E, T, L, K, \lambda, \kappa \rangle$ and an initial configuration $\langle v_0, A \rangle$ such that Player 1 wins from $\langle v, A \rangle$ in $\mathcal{G}$ iff w is accepted by $\mathcal{A}$. The vertices of $\mathcal{G}$ consist of *main* and *intermediate* vertices. Consider a configuration $c = \langle q, i, \langle \gamma_1, \ldots, \gamma_m \rangle \rangle$ of $\mathcal{A}$. We simulate c in $\mathcal{G}$ using $c' = \langle v, A \rangle$ as follows. First, the main vertices are $Q \times \{1, \ldots, m\} \times \Gamma$ and keep track of the control state and position on the tape. The main vertex that simulates $c = \langle q, i, \langle \gamma_1, \ldots, \gamma_m \rangle \rangle$ is $v = \langle q, i, \gamma_i \rangle$. We define $v \in V_i$ iff $q \in Q_i$. Second, we use locks to keep track of the tape contents. For each $1 \leq i \leq m$ and $\gamma \in \Gamma$, we introduce a lock $\ell_{i,\gamma}$. Then, in the configuration $c' = \langle v, A \rangle$ that simulates c, the only locks that are open are ℓ_{i,γ_i}, for $i \in \{1, \ldots, m\}$. Next, we describe the transitions, where intermediate vertices are used for book-keeping. The neighbors of a main vertex v are the intermediate vertices $\{\langle v, t \rangle : t \in \delta(q, \gamma)\}$, where a transition of $\mathcal{A}$ is $t = \langle q', \gamma', B \rangle$, meaning that the next control state is q', the tape head moves to $i+1$ if $B = R$ and to $i-1$ if $B = L$, and the i-th tape content changes from γ to γ'. We update the state of the locks so that they reflect the tape contents: for the edge $\langle v, \langle v, t \rangle \rangle$, we have $\kappa(\langle v, \langle v, t \rangle \rangle) = \{k_{i,\gamma}, k_{i,\gamma'}\}$.

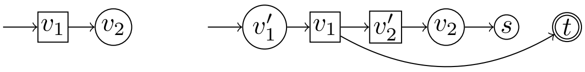


Figure 2 From turn-based to optional-grabbing games.

That is, traversing the edge turn the keys to close $\ell_{i,\gamma}$ and open $\ell_{i,\gamma'}$. The neighbors of $\langle v, t\rangle$ are main vertices having control state q' and head position i'. Recall that the third component of a main vertex is the tape content at the current position. We use the locks' state to prevent moving to main vertices with incorrect tape content: outgoing edges from $\langle v, t\rangle$ are of the form $\langle\langle v, t\rangle, \langle q', i', \gamma''\rangle\rangle$ and is labeled by the lock $\ell_{i',\gamma''}$. That is, the edge can only be traversed when the i'-th tape position is γ''. It is not hard to verify that there is a one-to-one correspondence between runs of $\mathcal{A}$ and plays of $\mathcal{G}$. Thus, Player 1 forces $\mathcal{A}$ to reach a configuration with control state q_{acc} iff Player 1 forces to reach a main vertex with control state q_{acc}. Note that the construction is clearly polynomial since $\mathcal{G}$ has $|Q| \cdot m \cdot |\Gamma|$ main vertices. ◀

3.2.1 From Lock & Key games to optional-grabbing pawn games

Throughout this section, fix a Lock & Key game $\mathcal{G}$ and an initial configuration c. We construct an optional-grabbing pawn game $\mathcal{G}'$ over a set of pawns $[d]$, and identify an initial configuration c' such that Player 1 wins in $\mathcal{G}$ from c iff Player 1 wins from c' in $\mathcal{G}'$.

From turn-based games to optional-grabbing games

In this section, we consider the case in which $\mathcal{G}$ has no keys or locks, thus $\mathcal{G}$ is a turn-based game. The reduction is depicted in Fig. 2. Denote the turn-based game $\mathcal{G} = \langle V, E, T\rangle$ with $V = V_1 \cup V_2$ and initial vertex v_0. We construct an OVPP optional-grabbing $\mathcal{G}' = \langle V', E', T'\rangle$, where $V' = V \cup \{v' : v \in V\} \cup \{s, t\}$. We add edges to ensure that the player who owns a vertex $v \in V$ is the player who moves from v in $\mathcal{G}'$: we have $\langle v', v\rangle \in E'$, and if $v \in V_1$, then $\langle v, s\rangle \in E'$, and if $v \in V_2$, then $\langle v, t\rangle \in E'$. We redirect each edge $\langle u, v\rangle$ in $\mathcal{G}$ to $\langle u, v'\rangle$ in $\mathcal{G}'$. Intuitively, for $v \in V_1$, a Player 1 winning strategy will guarantee that v' is always in the control of Player 2, and following her move at v', Player 1 must grab v otherwise Player 2 wins and choose the next location. And dually for $v \in V_2$. Let $V'_1 = V_1 \cup \{v' : v \in V_2\}$, the initial configuration of $\mathcal{G}'$ is $\langle v_0, V'_1\rangle$, that is Player 2 controls $V_2 \cup \{v' : v \in V_1\}$. Formally, we prove the following in the full version.

▶ **Lemma 9.** *For a turn-based game $\mathcal{G}$, Player 1 wins $\mathcal{G}$ from a vertex $v_0 \in V$ iff Player 1 wins the optional-grabbing game $\mathcal{G}'$ from configuration $\langle v_0, V'_1\rangle$.*

Gadgets for simulating locks and keys

For each lock $\ell \in L$ and its corresponding key $k \in K$, we construct gadgets $\mathcal{G}_\ell$ and $\mathcal{G}_k$ that simulate the operations of ℓ and k in $\mathcal{G}'$. The gadgets in two *states* are depicted in Fig. 3. We highlight three pawns colored blue, green, and red, respectively owning, $\{v_1^\ell, v_1^k\}$, $\{v_2^\ell, v_2^k, v_7^k, v_8^k\}$, and $\{v_{\text{in}}^k, v_4^k, v_5^k, v_6^k\}$. Each of the other vertices (colored white) is owned by a fresh pawn. Intuitively, for each lock ℓ, we identify two sets $\mathcal{P}_O^\ell, \mathcal{P}_C^\ell \subseteq 2^{[d]}$, respectively representing an open and closed state of ℓ. We will ensure that when entering and exiting a gadget, the configuration is in $\mathcal{P}_O^\ell \cup \mathcal{P}_C^\ell$. When the set of pawns that Player 1 controls is in $\mathcal{P}_O^\ell$ and $\mathcal{P}_C^\ell$, we respectively say that $\mathcal{G}_\ell$ is in open and closed state, and similarly for $\mathcal{G}_k$ as stated below. Formally, we define $\mathcal{P}_O^\ell = \{P \in 2^{[d]} : v_1^\ell \notin P \wedge v_2^\ell \in P\}$ and $\mathcal{P}_C^\ell = \{P \in 2^{[d]} : v_1^\ell \in P \wedge v_2^\ell \notin P\}$.

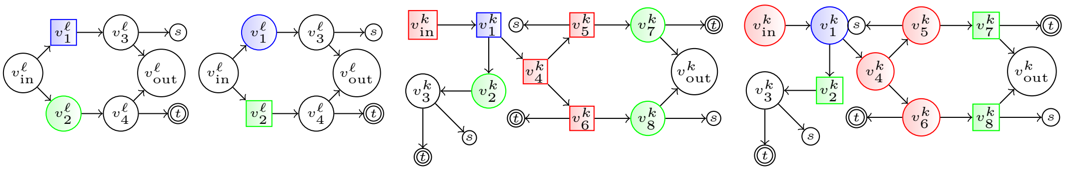


Figure 3 From left to right: $\mathcal{G}_\ell$ in open and closed state and $\mathcal{G}_k$ in open and closed state.

▶ **Lemma 10.** *Let $i \in \{1,2\}$. An open lock stays open: If Player i enters $\mathcal{G}_\ell$ in $\mathcal{P}_O^\ell$, then he has a strategy that guarantees that either he wins $\mathcal{G}'$ or $\mathcal{G}_\ell$ is exited in $\mathcal{P}_O^\ell$. A closed lock cannot be crossed: If Player i enters $\mathcal{G}_\ell$ in $\mathcal{P}_C^\ell$, then Player $-i$ has a strategy that guarantees that Player i loses $\mathcal{G}'$.*

Proof. We prove for Player 1 and the proof is dual for Player 2. First, suppose Player 1 enters $\mathcal{G}_\ell$ in $\mathcal{P}_O^\ell$. Player 2 may or may not grab v_{in}^ℓ, and the game can proceed to either v_1^ℓ or v_2^ℓ. We argue that if the game proceeds to v_1^ℓ, then Player 1 will not grab v_1^ℓ. We can also similarly show that if the game proceeds to v_2^ℓ, then Player 2 will not grab v_2^ℓ. Player 2 controls v_1^ℓ. We claim that if Player 1 grabs v_1^ℓ, he will lose the game. Indeed, following Player 1's move in v_1^ℓ, Player 2 will grab v_3^ℓ and move the token to the sink vertex s to win the game. Thus, Player 1 does not grab v_1^ℓ and keeps it in the control of Player 2. Following Player 2's move in v_1^ℓ, Player 1 grabs v_3^ℓ and proceeds to exit $\mathcal{G}_\ell$. Note that when $\mathcal{G}_\ell$ is exited, Player 1 maintains control of v_2^ℓ and Player 2 maintains control of v_1^ℓ, thus the configuration is in $\mathcal{P}_O^\ell$. Second, suppose that Player 1 enters $\mathcal{G}_\ell$ in $\mathcal{P}_C^\ell$. Then, Player 2 grabs v_{in}^ℓ and moves the token to v_1^ℓ. Since Player 1 controls v_1^ℓ he must make the next move. Player 2 then grabs v_3^ℓ and moves the token to s to win the game. ◀

Next, we present the gadget $\mathcal{G}_k$ for simulating the operation of a key k (see Fig. 3). Intuitively, we maintain that $\mathcal{G}_k$ is in open state iff $\mathcal{G}_\ell$ is in open state, and traversing $\mathcal{G}_k$ swaps the state of both. We define sets of configurations $\mathcal{P}_O^k = \{P \in 2^{[d]} : \{v_{\text{in}}^k, v_1^k, v_4^k, v_5^k, v_6^k\} \cap P = \emptyset \wedge \{v_2^k, v_7^k, v_8^k\} \subseteq P\}$ and $\mathcal{P}_C^k = \{P \in 2^{[d]} : \{v_{\text{in}}^k, v_1^k, v_4^k, v_5^k, v_6^k\} \subseteq P \wedge \{v_2^k, v_7^k, v_8^k\} \cap P = \emptyset\}$ (see Fig. 3). Note that $\mathcal{P}_O^k \subseteq \mathcal{P}_O^\ell$ and $\mathcal{P}_C^k \subseteq \mathcal{P}_C^\ell$ since v_i^k and v_i^ℓ are owned by the same pawn for $i \in [2]$. In the full version, we prove the following.

▶ **Lemma 11.** *Turning k closes an open ℓ: Let $i \in \{1,2\}$. If Player i enters $\mathcal{G}_k$ in $\mathcal{P}_O^k$, then he has a strategy that ensures that either Player i wins $\mathcal{G}'$ or $\mathcal{G}_k$ is exited in $\mathcal{P}_C^k$. Turning k opens a closed ℓ: when Player i enters $\mathcal{G}_k$ in $\mathcal{P}_C^k$, Player i ensures that either he wins $\mathcal{G}'$ or $\mathcal{G}_k$ is exited in $\mathcal{P}_O^k$.*

Putting it all together

We describe the construction of a pawn game $\mathcal{G}'$ from a Lock & Key game $\mathcal{G}$. We assume w.l.o.g. that each edge $\langle u, v\rangle$ in $\mathcal{G}$ is labeled by at most one lock or key since an edge that is labeled by multiple locks or keys can be split into a chain of edges, each labeled by a single lock or a key. We describe the construction of $\mathcal{G}'$. We first apply the construction for turn-based games on $\mathcal{G}$ while "ignoring" the locks and keys. Recall that the construction introduces fresh vertices so that an edge $e = \langle u, v\rangle$ in $\mathcal{G}$ is mapped to an edge $e' = \langle u', v\rangle$ in $\mathcal{G}'$. We re-introduce the locks and keys so that the labeling of e' coincides with the labeling of e. Next, we replace an edge e' that is labeled by a lock ℓ, by a copy of $\mathcal{G}_\ell$, and if e is labeled by a key k, we replace e' by a copy of $\mathcal{G}_k$. Note that multiple edges could be labeled by the same lock ℓ. In such a case we use fresh vertices in each copy of $\mathcal{G}_\ell$, but crucially, all gadgets

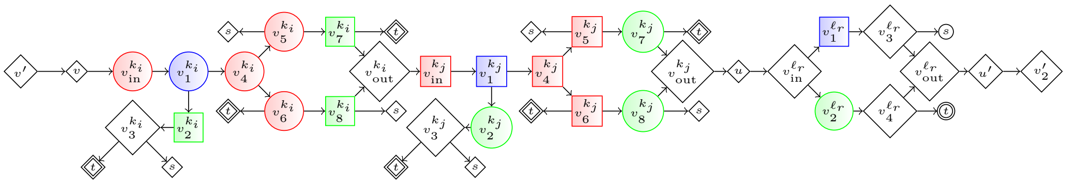


Figure 4 A δ-path is a path between two primed main vertices in an optional- or always-grabbing game, and it crosses two key gadgets and one lock gadget.

share the same pawns so that they share the same state. And similarly for keys. For an illustration of this construction, see Fig. 4, which applies the construction on a Lock & Key game that is output from the reduction in Thm. 8.

Finally, given an initial configuration $c = \langle v, A\rangle$ of $\mathcal{G}$ we define an initial configuration $c' = \langle v, P\rangle$ of $\mathcal{G}'$. Note that the initial vertex is the entry point of the gadget that simulates v in $\mathcal{G}'$. For each lock ℓ and corresponding key k, if ℓ is open according to A, then $P \in \mathcal{P}^{\ell}_{O}$, i.e., both $\mathcal{G}_\ell$ and $\mathcal{G}_k$ are initially in open state. And similarly when ℓ is closed according to A. Combining the properties in Lemmas 9, 10, and 11 implies that Player 1 wins $\mathcal{G}$ from c iff Player 1 wins $\mathcal{G}'$ from c'. Thus, by Thm. 8, we have the following.

▶ **Theorem 12.** *MVPP optional-grabbing PAWN-GAMES is EXPTIME-complete.*

4 Always-Grabbing Pawn Games

In this section, we study always-grabbing pawn games and show that MVPP always-grabbing pawn-games are EXPTIME-complete. The main challenge is proving the lower bound. We proceed as follows. Let $\mathcal{M}$ be an ATM. Apply the reduction in Thm 8 and the one in Thm. 12 to obtain pairs $\langle \mathcal{G}, c\rangle$ and $\langle \mathcal{G}', c'\rangle$, where $\mathcal{G}$ and $\mathcal{G}'$ are respectively Lock & Key and optional-grabbing games with initial configurations c and c' respectively. We devise a construction that takes $\langle \mathcal{G}', c'\rangle$ and produces an always-grabbing game $\mathcal{G}''$ and a configuration c'' such that Player 1 wins from c'' in $\mathcal{G}''$ iff he wins from c in $\mathcal{G}$.

Our analysis heavily depends on the special structure of $\mathcal{G}'$. The construction in Thm. 8 outputs a game $\mathcal{G}$ with main vertices of the form $\langle q, i, \gamma\rangle$ (q is a state, i is a tape position, and γ is a letter in the tape alphabet). A play of $\mathcal{G}$ can be partitioned into paths between main vertices. Each such path corresponds to one transition of the Turing machine and traverses two keys and a lock before again reaching a main vertex. Recall that when constructing $\mathcal{G}'$ from $\mathcal{G}$, we replace locks and keys with their respective gadgets, and for every vertex v that belongs to $\mathcal{G}$, we add a new primed vertex v' such that if v is controlled by Player i then v' is controlled by Player $-i$. We call a path in $\mathcal{G}'$ that corresponds to a path in $\mathcal{G}$ between two successive main vertices, say v and v', a *δ-path*. Fig. 4 depicts a δ-path. An important property of the specific optional-grabbing game $\mathcal{G}'$ that is constructed in Thm 8 from an ATM is that every play of $\mathcal{G}'$ consists of a sequence of δ-paths. More details on δ-paths can be found in the full version. The following observation can easily be verified:

▶ **Observation 13.** *A δ-path from v' to v'_2 consists of 20 turns.*

The following lemma is crucial for the construction of $\mathcal{G}''$.

▶ **Lemma 14.** *For $i \in \{1, 2\}$, if Player i has a strategy in the optional-grabbing game $\mathcal{G}'$ to cross a δ-path from v' to v'_2, then Player i has a strategy that moves the token in at least* 10 *rounds and Player $-i$ moves the token in at most 10 rounds in the δ-path.*

Let $\mathcal{G}' = \langle V', E', T' \rangle$ with d pawns. The game $\mathcal{G}''$ is constructed from $\mathcal{G}'$ by adding $2(d+10)$ fresh isolated vertices each owned by a fresh unique pawn. Formally, $\mathcal{G}'' = \langle V'', E', T' \rangle$, where $V'' = V' \cup \{v_1, v_2, \ldots, v_{2(d+10)}\}$ such that $v_j \notin V'$, for $j \in [2(d+10)]$. Consider a configuration $c' = \langle v, P \rangle$ in $\mathcal{G}'$. Let $c'' = \langle v, P \cup \{1, 2, \ldots, d+10\} \rangle$ be a configuration in $\mathcal{G}''$. Note that Lemma 14 also applies to the always-grabbing game $\mathcal{G}''$, and we get the following.

▶ **Corollary 15.** *For $i \in \{1,2\}$, if Player i has a strategy in the always-grabbing game $\mathcal{G}''$ to cross a δ-path from v' to v'_2, then Player i has a strategy such that out of the 20 rounds in the δ-path, the following hold.*

1. *Player $-i$* grabs *a pawn in at least 10 rounds, and*
2. *Player i* grabs *a pawn in at most 10 rounds.*

Corollary 15 follows directly from Lemma 14 since in an always-grabbing game, the number of times Player $-i$ grabs equals the number of times Player i moves. In the remaining part of this section, we show that Player 1 wins $\mathcal{G}'$ from c' iff Player 1 wins $\mathcal{G}''$ from the configuration c'' described above.

▶ **Lemma 16.** *For $i \in \{1,2\}$, Player i wins from c' in the optional-grabbing game $\mathcal{G}'$ iff he wins from c'' in the always-grabbing game $\mathcal{G}''$.*

Proof sketch. We prove that if Player 1 has a winning strategy f' in $\mathcal{G}'$ from c', then he has a winning strategy f'' from c'' in $\mathcal{G}''$. The case for Player 2 is analogous and the other direction follows from determinacy (Thm. 2). We construct f'' to mimic f' with the following difference. Whenever f' chooses not to grab, in order to follow the rules of the always-grabbing mechanism, f'' grabs a pawn owning an isolated vertex. This is possible since we show that we maintain the invariant that along a play in $\mathcal{G}''$ that consists of sequences of δ-paths, at the beginning of each δ-path, Player 2 has at least 10 isolated pawns. Note that the invariant holds initially due to the definition of c''. We show that it is maintained. Recall from the proof of Theorem 8 that crossing a δ-path simulates a transition in the Turing machine. Since *Player* 1 has a winning strategy in $\mathcal{G}'$, in a winning play, the strategy enables her to cross the δ-path. Thus, by Lem. 14, Player 1 moves in at least 10 rounds. Thus, Player 2 moves in at most 10 rounds, and during each such round, Player 1 grabs a pawn. Hence, Player 1 grabs at most 10 times which thus maintains the invariant. In the full version, we show that f'' is a winning Player 1 strategy. ◀

We now state the following theorem. While the lower bound follows from Thm. 12 and Lem. 16, the upper bound follows from Thm. 4.

▶ **Theorem 17.** *MVPP always-grabbing PAWN-GAMES is* EXPTIME*-complete.*

We conclude this section by adapting Thm. 5 to always-grabbing. Namely, we show that adding pawns to a player is never beneficial in MVPP always-grabbing games. (with the exception of the pawn that owns the current vertex). The proof can be found in the full version.

▶ **Theorem 18.** *Consider a configuration $\langle v, P \rangle$ of an MVPP always-grabbing pawn game $\mathcal{G}$. Let $j \in [d]$ such that $v \in V_j$ and $P' \subseteq P$. Assuming that $j \in P$ implies $j \in P'$, if Player 1 wins from $\langle v, P \rangle$, he wins from $\langle v, P' \rangle$. Assuming that $j \in \overline{P'}$ implies $j \in \overline{P}$, if Player 2 wins from $\langle v, P' \rangle$, she wins from $\langle v, P \rangle$.*

5 Always Grabbing-or-Giving Pawn Games

In this section, we show that MVPP always grabbing or giving games are in PTIME. We find it intriguing that a seemingly small change in the mechanism – allowing a choice between grabbing and giving instead of only grabbing – reduces the complexity to PTIME from EXPTIME-complete. We make the following simple observation.

▶ **Observation 19.** *In an always grabbing or giving game, every time Player i makes a move from a vertex v to a vertex u, Player $-i$ can decide which player controls u.*

If Player $-i$ does not control p_u that owns u and he wants to control u, he can grab p_u from Player i. If he does not want to control u and if he has p_u, he can give it to Player i.

Consider an always-grabbing-or-giving game $\mathcal{G} = \langle V, E, T\rangle$ and an initial configuration c. We construct a turn-based game $\mathcal{G}'$ and an initial vertex v_0 so that Player 1 wins in $\mathcal{G}$ from c iff he wins in $\mathcal{G}'$ from v_0. Let $\mathcal{G}' = \langle V', E', T'\rangle$, where $V' = \{\langle v, i\rangle, \langle \widehat{v}, i\rangle \mid v \in V, i \in \{1, 2\}\}$ with $V'_1 = \{\langle v, 1\rangle, \langle \widehat{v}, 1\rangle \mid v \in V\}$ and $V'_2 = \{\langle v, 2\rangle, \langle \widehat{v}, 2\rangle \mid v \in V\}$, $T' = T \times \{1, 2\}$, and $E' = \{(\langle v, i\rangle, \langle \widehat{u}, 3 - i\rangle), (\langle \widehat{u}, 3 - i\rangle, \langle u, i\rangle), (\langle \widehat{u}, 3 - i\rangle, \langle u, 3 - i\rangle) \mid (v, u) \in E, i \in \{1, 2\}\}$. We call each vertex $\langle v, i\rangle$ a *main* vertex and each $\langle \widehat{v}, i\rangle$ an *intermediate* vertex. Suppose that Player i moves the token from v to u in $\mathcal{G}$. If Player $-i$ decides to control u, then in $\mathcal{G}'$, the token moves from the main vertex $\langle v, i\rangle$ to the main vertex $\langle u, 3 - i\rangle$, else from $\langle v, i\rangle$ to the main vertex $\langle u, i\rangle$, and in each case, through the intermediate vertex $(\widehat{u}, 3 - i)$ that models the decision of Player $-i$ on the control of u. The target vertices T' are main vertices. The proof of the following lemma appears in the full version.

▶ **Lemma 20.** *Suppose Player 1 wins from configuration $\langle v, P\rangle$ in $\mathcal{G}$. If he controls v, he wins from $\langle v, 1\rangle$ in $\mathcal{G}'$, and if Player 2 controls v, Player 1 wins from $\langle v, 2\rangle$ in $\mathcal{G}'$. Dually, suppose that Player 2 wins from $\langle v, P\rangle$ in $\mathcal{G}$. If she controls v, then she wins from $\langle v, 2\rangle$ in $\mathcal{G}'$, and if Player 1 controls v, Player 2 wins from $\langle v, 1\rangle$ in $\mathcal{G}'$.*

Since the size of $\mathcal{G}'$ is polynomial in the size of $\mathcal{G}$, Thm. 2 implies the following.

▶ **Theorem 21.** *MVPP always-grab-or-give PAWN-GAMES is in PTIME.*

6 k-Grabbing Pawn Games

In this section, we consider pawn games under k-grabbing in increasing level of generality of the mechanisms. We start with positive news.

▶ **Theorem 22.** *OVPP k-grabbing PAWN-GAMES is in* PTIME*.*

Proof. Let $k \in \mathbb{N}$, an OVPP k-grabbing game $\mathcal{G} = \langle V, E, T\rangle$, and an initial configuration $c = \langle v_0, P_0\rangle$, where we refer to P_0 as a set of vertices rather than pawns. For a vertex $u \in V$, let $\eta(u)$ denote the minimum number of grabs with which Player 1 can guarantee winning $\mathcal{G}$ from configuration $\langle u, P_0\rangle$. The algorithm recursively computes η based on repeated calls to an algorithm to solve turn-based games (see Thm. 2).

For the base case, consider the turn-based game $\mathcal{G}_0 = \langle V, E, T\rangle$ with $V_1 = P_0$. Let $W^1_0 \subseteq V$ denote Player 1's winning region in $\mathcal{G}_0$. Clearly, for every $u \in W^1_0$, we have $\eta(v_0) = 0$, and for every $u \notin W^1_0$, we have $\eta(v_0) \geq 1$. For the inductive step, suppose that for $\ell \geq 0$, the set $W^1_\ell = \{u \in V : \eta(u) \leq \ell\}$ has been found. That is, for every $u \notin W^1_\ell$, Player 2 has a strategy that wins the ℓ-grabbing pawn game $\mathcal{G}$ from configuration $\langle u, P_0\rangle$. We show how to find $W^1_{\ell+1}$ in linear time. Let the *border* of W^1_ℓ, denoted B_ℓ, be the set of vertices from which W^1_ℓ can be reached in one step, thus $B_\ell = \{v \in V : v \notin W^1_\ell : N(v) \cap W^1_\ell \neq \emptyset\}$.

Note that the vertices in B_ℓ are all controlled by Player 2 since otherwise, such vertices will be in the set W^1_ℓ. In the full version, we show that a vertex $u \notin W^1_\ell$ has $\eta(u) = \ell + 1$ iff Player 1 can force the game from configuration $\langle u, P_0 \rangle$ to a vertex in B_ℓ in one or more rounds without making any grab. Player 1 wins from such a vertex u by forcing the game into B_ℓ, grabbing the pawn in B_ℓ, and proceeding to W_ℓ, where by the induction hypothesis, he wins with the remaining grabs. Computing $W^1_{\ell+1}$ roughly entails a solution to a turn-based game with target set $B_\ell \cup W^1_\ell$. ◀

The proof of the following theorem, which can be found in the full version, is obtained by a reduction from SET-COVER.

▶ **Theorem 23.** *MVPP k-grabbing game PAWN-GAMES is NP-hard.*

We conclude this section by studying OMVPP games.

▶ **Lemma 24.** *OMVPP k-grabbing PAWN-GAMES is PSPACE-hard.*

Proof. Consider an input $\phi = Q_1 x_1 \ldots Q_n x_n C_1 \wedge \ldots \wedge C_m$ to TQBF, where $Q_i \in \{\exists, \forall\}$, for $1 \leq i \leq n$, each C_j, for $1 \leq j \leq m$, is a clause over the variables $x_1, \ldots, x_n$. We construct an OMVPP n-grabbing pawn game $\mathcal{G} = \langle V, E, T \rangle$ such that Player 1 wins iff ϕ is true. We describe the intuition and the details can be found in the full version. The structure of $\mathcal{G}$ is chain-like. Player 1 needs to cross the chain in order to win. The first part of the chain requires Player 1 to grab, for each variable x_i, either a pawn p_i or a pawn $\neg p_i$. For existentially-quantified variables, Player 1 decides which of the two is grabbed, and for universally-quantified variables, Player 2 decides. In the second part of $\mathcal{G}$, we verify that the corresponding assignment is valid. Certain positions of $\mathcal{G}$ correspond to a clause C_j, for $j \in [m]$, which Player 1 must take control over during the first part of $\mathcal{G}$. The key is to use OMVPP: We define that if x_i appears in C_j, then p_i is an owner of C_j, and if $\neg x_i$ appears in C_j, then $\neg p_i$ is an owner of C_j. Thus, Player 1 controls C_j iff he grabbed a pawn that owns C_j which is iff the assignment satisfies C_j. ◀

We turn to study the upper bound. The following lemma bounds the provides a polynomial bound on the length of a winning play for Player 1. The core of the proof, which can be found in the full version, intuitively shows that we can restrict attention to Player 1 strategies that grab at least once in a sequence of $|V|$ rounds. Otherwise, the game enters a cycle that is winning for Player 2.

▶ **Lemma 25.** *Consider an OMVPP k-grabbing PAWN-GAME $\mathcal{G} = \langle V, E, T \rangle$, and an initial configuration c that is winning for Player 1. Then, Player 1 has a strategy such that, for every Player 2 strategy, a target in T is reached within $|V| \cdot (k+1)$ rounds.*

For the upper bound, in the full version, we describe an algorithm performing a depth-first traversal of the configuration graph of a game while storing, at a time, only a branch in PSPACE. By Lem. 25, each branch of such a traversal has polynomial length, leading to the PSPACE upper bound. We thus have the following.

▶ **Theorem 26.** *OMVPP k-grabbing PAWN-GAMES is PSPACE-complete.*

7 Discussion

We introduce pawn games, a class of two-player turn-based games in which control of vertices changes dynamically throughout the game. Pawn games constitute a class of succinctly-represented turn-based games. We identify natural classes that are in PTIME.

Our EXPTIME-hardness results are based on Lock & Key games, which we hope will serve as a framework for proving lower bounds. We mention directions for future research. First, we leave several open problems; e.g., for MVPP k-grabbing pawn games, we only show NP-hardness and membership in PSPACE. Second, we focused on reachability games. It is interesting to study pawn games with richer objectives such as parity or quantitative objectives. Third, it is interesting to consider other pawn-transferring mechanisms and to identify properties of mechanisms that admit low-complexity results. Finally, grabbing pawns is a general concept and can be applied to more involved games like stochastic or concurrent games.

References

1 M. Aghajohari, G. Avni, and T. A. Henzinger. Determinacy in discrete-bidding infinite-duration games. *Log. Methods Comput. Sci.*, 17(1), 2021.

2 P. Alizadeh Alamdari, G. Avni, T. A. Henzinger, and A. Lukina. Formal methods with a touch of magic. In *Proc. 20th FMCAD*, 2020.

3 S. Almagor, G. Avni, and O. Kupferman. Repairing multi-player games. In *Proc. 26th CONCUR*, pages 325–339, 2015.

4 R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

5 K.R. Apt and E. Grädel. *Lectures in Game Theory for Computer Scientists*. Cambridge University Press, 2011.

6 G. Avni, R. Bloem, K. Chatterjee, T. A. Henzinger, B. Könighofer, and S. Pranger. Run-time optimization for learned controllers through quantitative games. In *Proc. 31st CAV*, pages 630–649, 2019.

7 G. Avni, P. Ghorpade, and S. Guha. A game of pawns. *CoRR*, abs/2305.04096, 2023. arXiv:2305.04096.

8 G. Avni, T. A. Henzinger, and V. Chonev. Infinite-duration bidding games. *J. ACM*, 66(4):31:1–31:29, 2019.

9 G. Avni and S. Sadhukhan. Computing threshold budgets in discrete-bidding games. In *Proc. 42nd FSTTCS*, volume 250 of *LIPIcs*, pages 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

10 G. Bielous and O. Kupferman. Coverage and vacuity in network formation games. In *Proc. 28th CSL*, 2020.

11 R. Brenguier, L. Clemente, P. Hunter, G. A. Pérez, M. Randour, J.-F. Raskin, O. Sankur, and M. Sassolas. Non-zero sum games for reactive synthesis. In *Proc. 10th LATA*, pages 3–23, 2016.

12 L. Brice, J.-F. Raskin, and M. van den Bogaard. Subgame-perfect equilibria in mean-payoff games. In *In Proc. 32nd CONCUR*, volume 203 of *LIPIcs*, pages 8:1–8:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

13 D. Busatto-Gaston, D. Chakraborty, S. Guha, G. A. Pérez, and J-F. Raskin. Safe learning for near-optimal scheduling. In *QEST, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 2021.

14 P. Cerný, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. *Formal Methods Syst. Des.*, 50(2-3):97–139, 2017.

15 K. Chatterjee, T. A. Henzinger, and N. Piterman. Strategy logic. *Inf. Comput.*, 208(6):677–693, 2010.

16 M. Develin and S. Payne. Discrete bidding games. *The Electronic Journal of Combinatorics*, 17(1):R85, 2010.

17 D. Fisman, O. Kupferman, and Y. Lustig. Rational synthesis. In *Proc. 16th TACAS*, pages 190–204, 2010.

18 E. Grädel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

19 B. Könighofer, M. Alshiekh, R. Bloem, L. Humphrey, R. Könighofer, U. Topcu, and C. Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, 2017.

20 O. Kupferman, G. Perelli, and M. Y. Vardi. Synthesis with rational environments. *Ann. Math. Artif. Intell.*, 78(1):3–20, 2016.

21 L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

22 A. J. Lazarus, D. E. Loeb, J. G. Propp, W. R. Stromquist, and D. H. Ullman. Combinatorial games under auction play. *Games and Economic Behavior*, 27(2):229–264, 1999.

23 F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi. Reasoning about strategies: On the model-checking problem. *ACM Trans. Comput. Log.*, 15(4):34:1–34:47, 2014.

24 F. Mogavero, A. Murano, and M. Y. Vardi. Reasoning about strategies. In *Proc. 30th FSTTCS*, pages 133–144, 2010.

25 G. Perelli. Enforcing equilibria in multi-agent systems. In *In Proc. 18th AAMAS*, pages 188–196. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

26 A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th POPL*, pages 179–190, 1989.

27 S. A. Seshia, N. Sharygina, and S. Tripakis. Modeling for verification. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 75–105. Springer, 2018.

28 Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, 2013.

29 R. S. Sutton and A. G. Barto. *Reinforcement learning – An introduction*. Adaptive computation and machine learning. MIT Press, 1998.

30 M. Ummels. *Stochastic multiplayer games: theory and algorithms*. PhD thesis, RWTH Aachen University, 2011.

31 J. van Benthem. An essay on sabotage and obstruction. In *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, pages 268–276, 2005.

32 M. J. Wooldridge, J. Gutierrez, P. Harrenstein, E. Marchioni, G. Perelli, and A. Toumi. Rational verification: From model checking to equilibrium checking. In *Proc. of the 30th AAAI*, pages 4184–4191, 2016.

Safety and Liveness of Quantitative Automata

Udi Boker
Reichman University, Herzliya, Israel

Thomas A. Henzinger
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Nicolas Mazzocchi
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

N. Ege Saraç
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Abstract

The safety-liveness dichotomy is a fundamental concept in formal languages which plays a key role in verification. Recently, this dichotomy has been lifted to quantitative properties, which are arbitrary functions from infinite words to partially-ordered domains. We look into harnessing the dichotomy for the specific classes of quantitative properties expressed by quantitative automata. These automata contain finitely many states and rational-valued transition weights, and their common value functions Inf, Sup, LimInf, LimSup, LimInfAvg, LimSupAvg, and DSum map infinite words into the totally-ordered domain of real numbers. In this automata-theoretic setting, we establish a connection between quantitative safety and topological continuity and provide an alternative characterization of quantitative safety and liveness in terms of their boolean counterparts. For all common value functions, we show how the safety closure of a quantitative automaton can be constructed in PTime, and we provide PSpace-complete checks of whether a given quantitative automaton is safe or live, with the exception of LimInfAvg and LimSupAvg automata, for which the safety check is in ExpSpace. Moreover, for deterministic Sup, LimInf, and LimSup automata, we give PTime decompositions into safe and live automata. These decompositions enable the separation of techniques for safety and liveness verification for quantitative specifications.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Theory of computation → Quantitative automata

Keywords and phrases quantitative safety, quantitative liveness, quantitative automata

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.17

Related Version *Full Version*: `https://arxiv.org/abs/2307.06016`

Funding This work was supported in part by the ERC-2020-AdG 101020093 and the Israel Science Foundation grant 2410/22.

Acknowledgements We thank Christof Löding for pointing us to some results on PSpace-hardess of universality problems and the anonymous reviewers for their helpful comments.

1 Introduction

Safety and liveness [2] are fundamental concepts in the specification of system behaviors and their verification. While safety characterizes whether a system property can *always* be falsified by a finite prefix of its violating executions, liveness characterizes whether this is *never* possible. A celebrated result shows that *every* property is the intersection of a safety property and a liveness property [2]. This decomposition significantly impacts verification efforts: every verification task can be split into verifying a safety property, which can be solved by lighter methods, such as computational induction, and a liveness property, which requires heavier methods, such as ranking functions.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 17; pp. 17:1–17:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

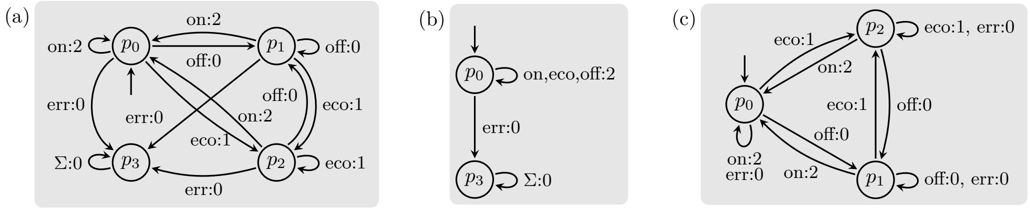


Figure 1 (a) A LimSup-automaton $\mathcal{A}$ modeling the long-term maximal power consumption of a device. **(b)** An Inf-automaton (or a LimSup-automaton) expressing the safety closure of $\mathcal{A}$. **(c)** A LimSup-automaton expressing the liveness component of the decomposition of $\mathcal{A}$.

The notions of safety and liveness consider system properties in full generality: every set of system executions – even the uncomputable ones – can be seen through the lens of the safety-liveness dichotomy. To bring these notions more in line with practical requirements, their projections onto formalisms with desirable closure and decidability properties, such as ω-regular languages, have been studied thoroughly. For example, [3] gives a construction for the safety closure of a Büchi automaton and shows that Büchi automata are closed under the safety-liveness decomposition. In turn, [29] describes an efficient model-checking algorithm for Büchi automata that define safety properties.

Boolean properties define *sets* of system executions or, equivalently, characteristic functions mapping each infinite execution to a binary truth value. Quantitative properties [10] generalize their boolean counterparts; they are *functions* from infinite executions to richer *value domains*, such as the real numbers, allowing the specification and verification of system properties not only for correctness but also for performance and robustness.

As in the boolean case, quantitative extensions of safety and liveness [25] have been defined through the falsifiability, from finite execution prefixes, of quantitative membership hypotheses, which are claims that a given value is a lower or upper bound on the values of certain executions. In particular, quantitative safety (resp. co-safety) characterizes whether every wrong lower (resp. upper) bound hypothesis can *always* be rejected by a finite execution prefix, and quantitative liveness (resp. co-liveness) characterizes whether some wrong lower (resp. upper) bound hypothesis can *never* be rejected by a finite execution prefix. In this setting, the safety closure of a quantitative property maps each execution to the greatest lower bound over the best values that all execution prefixes can have via some continuations; in other words, it is the least safety property that bounds the given property from above [25].

Let us give some examples. Suppose we have three observations `on`, `eco`, and `off`, corresponding to the operational modes of a device, with the power consumption values 2, 1, and 0, respectively. The quantitative property MinPow maps every execution to the minimum among the power consumption values of the modes that occur in the execution, and MaxPow maps them to the corresponding maximum. The property MinPow is safe because, for every execution and power consumption value v, if the MinPow value of the execution is less than v, then there is a finite prefix of the execution in which an operational mode with a power value less than v occurs, and after this prefix, no matter what infinite execution follows, MinPow value cannot be greater. The property MaxPow is live because, for every execution (whose MaxPow value is not the maximal possible value of 2), there is a power value v such that the MaxPow value of the execution is less than v, but for all of its finite prefixes there is an infinite continuation that achieves a MaxPow value of at least v.

Similarly to how boolean automata (e.g., regular and ω-regular automata) define classes of boolean properties amenable to boolean verification, quantitative automata (e.g., limit-average and discounted-sum automata) define classes of quantitative properties amenable to quantitative verification. Quantitative automata generalize standard boolean automata with weighted transitions and a value function that accumulates an infinite sequence of weights into a single value, a generalization of acceptance conditions of ω-regular automata. Let us extend the set of possible observations in the above example with err, which denotes an error in the device. In Figure 1a, we describe a quantitative automaton using the value function LimSup to express the long-term maximal power consumption of the device.

In this work, we study the projection of the quantitative safety-liveness dichotomy onto the properties definable by common quantitative automata. First, we show how certain attributes of quantitative automata simplify the notions of safety and liveness. Then, we use these simplifications to the study safety and liveness of the classes of quantitative automata with the value functions Inf, Sup, LimInf, LimSup, LimInfAvg, LimSupAvg, and DSum [10].

In contrast to general quantitative properties, these quantitative automata use functions on the totally-ordered domain of the real numbers (as opposed to a more general partially-ordered domain). In addition, quantitative automata have the restriction that only finitely many weights (those on the automaton transitions) can contribute to the value of an execution. These constraints allow us to provide alternative, simpler characterizations of safety for properties defined by quantitative automata. In particular, we show that, for totally-ordered value domains, a quantitative property is safe iff, for every value v, the set of executions whose value is at least v is safe in the boolean sense. The total-order restriction also allows us to study quantitative safety through the lens of topological continuity. In particular, we characterize safety properties as continuous functions with respect to the left-order topology of their totally-ordered value domain. Moreover, we define the safety of value functions and show that a value function is safe iff every quantitative automaton equipped with this value function expresses a safety property. For example, Inf is a safe value function. Pushing further, we characterize discounting properties and value functions as those that are uniformly continuous and show that it characterizes the conjunction of safety and co-safety. For example, DSum is a discounting value function, therefore both safe and co-safe.

We prove that the considered classes of quantitative automata have the ability to express the least upper bound over their values, namely, they are supremum-closed. Similarly as for safety and the total-order constraint, this ability helps us simplify quantitative liveness. For supremum-closed quantitative properties, we show that a property is live iff for every value v, the set of executions whose value is at least v is live in the boolean sense.

These simplifying characterizations of safety and liveness for quantitative automata prove useful for checking the safety and liveness of these automata, for constructing the safety closure of an automaton, and for decomposing an automaton into safety and liveness components. Let us recall the quantitative automaton in Figure 1a. Since it is supremum-closed, we can construct its safety closure in PTime by computing the maximal value it can achieve from each state. The safety closure of this automaton is shown in Figure 1b. For the value functions Inf, Sup, LimInf, LimSup, LimInfAvg, and LimSupAvg, the safety closure of a given automaton is an Inf-automaton, while for DSum, it is a DSum-automaton.

Evidently, one can check if a quantitative automaton $\mathcal{A}$ is safe by checking if it is equivalent to its safety closure, i.e., if $\mathcal{A}(w) = \mathit{SafetyCl}(\mathcal{A})(w)$ for every execution w. This allows for a PSpace procedure for checking the safety of Sup-, LimInf-, and LimSup-automata [10], but not for LimInfAvg- and LimSupAvg-automata, whose equivalence check is undecidable [15]. For these cases, we use the special structure of the safety-closure automaton for reducing safety checking to the problem of whether some other automaton expresses a constant function. We show that the latter problem is PSpace-complete for LimInfAvg- and LimSupAvg-automata, by a somewhat involved reduction to the limitedness problem of distance automata, and obtain an ExpSpace decision procedure for their safety check.

Thanks to our alternative characterization of liveness, one can check if a quantitative automaton $\mathcal{A}$ is live by checking if its safety closure is universal with respect to its maximal value, i.e., if $SafetyCl(\mathcal{A})(w) \geq \top$ for every execution w, where $\top$ is the supremum over the values of $\mathcal{A}$. For all value functions we consider except DSum, the safety closure is an Inf-automaton, which allows for a PSpace solution to liveness checking [10], which we show to be optimal. Yet, it is not applicable for DSum automata, as the decidability of their universality check is an open problem. Nonetheless, as we consider only universality with respect to the maximal value of the automaton, we can reduce the problem again to checking whether an automaton defines a constant function, which we show to be in PSpace for DSum-automata. This yields a PSpace-complete solution to the liveness check of DSum-automata.

Finally, we investigate the safety-liveness decomposition for quantitative automata. Recall the automaton from Figure 1a and its safety closure from Figure 1b. The liveness component of the corresponding decomposition is shown in Figure 1c. Intuitively, it ignores `err` and provides information on the power consumption as if the device never fails. Then, for every execution w, the value of the original automaton on w is the minimum of the values of its safety closure and the liveness component on w. Since we identified the value functions Inf and DSum as safe, their safety-liveness decomposition is trivial. For deterministic Sup-, LimInf-, and LimSup-automata, we provide PTime decompositions, where for Sup and LimInf it extends to nondeterministic automata at the cost of exponential determinization.

We note that our alternative, simpler characterizations of safety and liveness of quantitative properties extend to co-safety and co-liveness. Our results for the specific automata classes are summarized in Table 1. While we focus on automata that resolve nondeterminism by sup, their duals hold for quantitative co-safety and co-liveness of automata that resolve nondeterminism by inf, as well as for deterministic automata. We leave the questions of co-safety and co-liveness for automata that resolve nondeterminism by sup open.

Related Work. The notions of safety and liveness for boolean properties were first presented in [30] and were later formally defined in [2]. The projections of safety and liveness onto properties definable by Büchi automata were studied in [3]. For linear temporal logic, safety and liveness were studied in [37], where checking whether a given formula is safe was shown to be PSpace-complete. The safety-liveness dichotomy also shaped various efforts on verification, such as an efficient model-checking algorithm for safe Büchi automata [29]. A framework for monitorability through the lens of safety and liveness was given in [34], and a monitor model for safety properties beyond ω-regular ones was defined and studied in [18].

Quantitative properties (a.k.a. quantitative languages [10]) generalize their boolean counterparts by moving from a binary domain of truth values to richer value domains such as the real numbers. In the past decades, quantitative properties and automata have been studied extensively in games with quantitative objectives [6, 9], specification and analysis of system robustness [33], measuring the distance between two systems or specifications [13, 23], best-effort synthesis and repair [5, 12], approximate monitoring [26, 24], and more [11, 8, 17].

Safety and liveness of general quantitative properties were defined and studied in [25]. In particular, quantitative safety properties were characterized as upper semicontinuous functions, and every quantitative property was shown to be the pointwise minimum of a safety property and a liveness property. Yet, these definitions have not been studied from the perspective of quantitative finite-state automata.

Other definitions of safety and liveness for nonboolean formalisms were presented in [32, 20]. While [32] focuses on multi-valued formalisms with the aim of providing model-checking algorithms, [20] focuses on the monitorability view of safety and liveness in richer value

Table 1 The complexity of performing the operations on the left column with respect to nondeterministic automata with the value function specified on the top row.

	Inf	Sup, LimInf, LimSup	LimInfAvg, LimSupAvg	DSum
Constructing *SafetyCl*($\mathcal{A}$)	$O(1)$	PTime Theorem 4.18		$O(1)$
Constant-function check	PSpace-complete Proposition 3.2 and Theorems 3.3 and 3.7			
Safety check	$O(1)$	PSpace-complete Theorem 4.22	ExpSpace; PSpace-hard Theorem 4.23 and Lemma 4.21	$O(1)$
Liveness check	PSpace-complete Theorem 5.9			
Safety-liveness decomposition	$O(1)$	PTime if deterministic Theorems 5.10 and 5.11	Open	$O(1)$

domains. The relations between these definitions were investigated in [25]. Notably, a notion of safety was studied for the rational-valued min-plus weighted automata on finite words in [38]. They take a weighted property as v-safe for a given rational v when for every execution w, if the hypothesis that the value of w is strictly less than v is wrong (i.e., its value is at least v), then there is a finite prefix of w to witness it. Then, a weighted property is safe when it is v-safe for *some* value v. Given a nondeterministic weighted automaton $\mathcal{A}$ and an integer v, they show that it is undecidable to check whether $\mathcal{A}$ is v-safe. In contrast, the definition in [25], which we follow, quantifies over *all* values and non-strict lower-bound hypotheses. Moreover, for this definition, we show that checking safety of all common classes of quantitative automata is decidable, even in the presence of nondeterminism. Finally, [4] studies the safety and co-safety of discounted-sum comparator automata. While these automata internally use discounted summation, they are boolean automata recognizing languages, and therefore they only consider boolean safety and co-safety.

Our study shows that determining whether a given quantitative automaton expresses a constant function is a key for deciding safety and liveness, in particular for automata classes in which equivalence or universality checks are undecidable. To the best of our knowledge, this problem has not been studied before.

2 Quantitative Properties and Automata

Let $\Sigma = \{a, b, \ldots\}$ be a finite alphabet of letters (observations). An infinite (resp. finite) word is an infinite (resp. finite) sequence of letters $w \in \Sigma^\omega$ (resp. $u \in \Sigma^*$). For a natural number $n \in \mathbb{N}$, we denote by Σ^n the set of finite words of length n. Given $u \in \Sigma^*$ and $w \in \Sigma^* \cup \Sigma^\omega$, we write $u \prec w$ (resp. $u \preceq w$) when u is a strict (resp. nonstrict) prefix of w. We denote by $|w|$ the length of $w \in \Sigma^* \cup \Sigma^\omega$ and, given $a \in \Sigma$, by $|w|_a$ the number of occurrences of a in w. For $w \in \Sigma^* \cup \Sigma^\omega$ and $0 \leq i < |w|$, we denote by $w[i]$ the ith letter of w.

A *value domain* $\mathbb{D}$ is a poset. Unless otherwise stated, we assume that $\mathbb{D}$ is a nontrivial (i.e., $\bot \neq \top$) complete lattice. Whenever appropriate, we write 0 or $-\infty$ instead of $\bot$ for the least element, and 1 or ∞ instead of $\top$ for the greatest element. We respectively use the terms minimum and maximum for the greatest lower bound and the least upper bound of finitely many elements.

A *quantitative property* is a total function $\Phi : \Sigma^\omega \to \mathbb{D}$ from the set of infinite words to a value domain. A boolean property $P \subseteq \Sigma^\omega$ is a set of infinite words. We use the boolean domain $\mathbb{B} = \{0, 1\}$ with $0 < 1$ and, in place of P, its *characteristic property* $\Phi_P : \Sigma^\omega \to \mathbb{B}$, which is defined by $\Phi_P(w) = 1$ if $w \in P$, and $\Phi_P(w) = 0$ if $w \notin P$. When we say just *property*, we mean a quantitative one.

Given a property $\Phi : \Sigma^\omega \to \mathbb{D}$ and a value $v \in \mathbb{D}$, we define $\Phi_{\sim v} = \{w \in \Sigma^\omega \mid \Phi(w) \sim v\}$ for $\sim \in \{\leq, \geq, \nleq, \ngeq\}$. The *top value* of a property Φ is $\sup_{w\in\Sigma^\omega} \Phi(w)$, which we denote by $\top_\Phi$, or simply $\top$ when Φ is clear from the context.

A *nondeterministic quantitative*[1] *automaton* (or just automaton from here on) on words is a tuple $\mathcal{A} = (\Sigma, Q, \iota, \delta)$, where Σ is an alphabet; Q is a finite nonempty set of states; $\iota \in Q$ is an initial state; and $\delta \colon Q \times \Sigma \to 2^{(\mathbb{Q}\times Q)}$ is a finite transition function over weight-state pairs. A *transition* is a tuple $(q, \sigma, x, q') \in Q \times \Sigma \times \mathbb{Q} \times Q$, such that $(x, q') \in \delta(q, \sigma)$, also written $q \xrightarrow{\sigma:x} q'$. (There might be finitely many transitions with different weights over the same letter between the same states.[2]) We write $\gamma(t) = x$ for the weight of a transition $t = (q, \sigma, x, q')$. $\mathcal{A}$ is deterministic if for all $q \in Q$ and $a \in \Sigma$, the set $\delta(q, a)$ is a singleton. We require the automaton $\mathcal{A}$ to be *total*, namely that for every state $q \in Q$ and letter $\sigma \in \Sigma$, there is at least one state q' and a transition $q \xrightarrow{\sigma:x} q'$. For a state $q \in Q$, we denote by $\mathcal{A}^q$ the automaton that is derived from $\mathcal{A}$ by setting its initial state ι to q.

A run of $\mathcal{A}$ on a word w is a sequence $\rho = q_0 \xrightarrow{w[0]:x_0} q_1 \xrightarrow{w[1]:x_1} q_2 \dots$ of transitions where $q_0 = \iota$ and $(x_i, q_{i+1}) \in \delta(q_i, w[i])$. For $0 \leq i < |w|$, we denote the ith transition in ρ by $\rho[i]$, and the finite prefix of ρ up to and including the ith transition by $\rho[..i]$. As each transition t_i carries a weight $\gamma(t_i) \in \mathbb{Q}$, the sequence ρ provides a weight sequence $\gamma(\rho) = \gamma(t_0)\gamma(t_1)\dots$ A Val (e.g., DSum) automaton is one equipped with a value function $\mathsf{Val} : \mathbb{Q}^\omega \to \mathbb{R}$, which assigns real values to runs of $\mathcal{A}$. We assume that Val is bounded for every finite set of rationals, i.e., for every finite $V \subset \mathbb{Q}$ there exist $m, M \in \mathbb{R}$ such that $m \leq \mathsf{Val}(x) \leq M$ for every $x \in V^\omega$. Note that the finite set V corresponds to transition weights of a quantitative automaton, and the concrete value functions we consider satisfy this assumption.

The value of a run ρ is $\mathsf{Val}(\gamma(\rho))$. The value of a Val-automaton $\mathcal{A}$ on a word w, denoted $\mathcal{A}(w)$, is the supremum of $\mathsf{Val}(\rho)$ over all runs ρ of $\mathcal{A}$ on w. The *top value* of a Val-automaton $\mathcal{A}$ is the top value of the property it expresses, which we denote by $\top_\mathcal{A}$, or simply $\top$ when $\mathcal{A}$ is clear from the context. Note that when we speak of the top value of a property or an automaton, we always match its value domain to have the same top value.

Two automata $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent*, if they express the same function from words to reals. The size of an automaton consists of the maximum among the size of its alphabet, state-space, and transition-space, where weights are represented in binary.

We list below the value functions for quantitative automata that we will use, defined over infinite sequences $v_0 v_1 \dots$ of rational weights.

- $\mathsf{Inf}(v) = \inf\{v_n \mid n \geq 0\}$
- $\mathsf{Sup}(v) = \sup\{v_n \mid n \geq 0\}$
- $\mathsf{LimInf}(v) = \lim_{n\to\infty} \inf\{v_i \mid i \geq n\}$
- $\mathsf{LimSup}(v) = \lim_{n\to\infty} \sup\{v_i \mid i \geq n\}$
- $\mathsf{LimInfAvg}(v) = \mathsf{LimInf}\left(\frac{1}{n}\sum_{i=0}^{n-1} v_i\right)$
- $\mathsf{LimSupAvg}(v) = \mathsf{LimSup}\left(\frac{1}{n}\sum_{i=0}^{n-1} v_i\right)$
- For a discount factor $\lambda \in \mathbb{Q} \cap (0, 1)$, $\mathsf{DSum}_\lambda(v) = \sum_{i\geq 0} \lambda^i v_i$

Note that (i) when the discount factor $\lambda \in \mathbb{Q} \cap (0, 1)$ is unspecified, we write DSum, and (ii) $\mathsf{LimInfAvg}$ and $\mathsf{LimSupAvg}$ are also called $\underline{\mathsf{MeanPayoff}}$ and $\overline{\mathsf{MeanPayoff}}$ in the literature.

[1] We speak of "quantitative" rather than "weighted" automata, following the distinction made in [7] between the two.

[2] The flexibility of allowing "parallel" transitions with different weights is often omitted, as it is redundant for some value functions, including the ones we focus on in the sequel, while important for others.

The following statement allows us to consider Inf- and Sup-automata as only having runs with nonincreasing and nondecreasing, respectively, sequences of weights and to also consider them as LimInf- and LimSup-automata.

▶ **Proposition 2.1.** *Let* Val $\in \{\mathsf{Inf}, \mathsf{Sup}\}$*. Given a* Val*-automaton, we can construct in* PTIME *an equivalent* Val-, LimInf- *or* LimSup-*automaton whose runs yield monotonic weight sequences.*

Given a property Φ and a finite word $u \in \Sigma^*$, let $P_{\Phi,u} = \{\Phi(uw) \mid w \in \Sigma^\omega\}$. A property Φ is sup-*closed* (resp. inf-*closed*) when for every finite word $u \in \Sigma^*$ we have that $\sup P_{\Phi,u} \in P_{\Phi,u}$ (resp. $\inf P_{\Phi,u} \in P_{\Phi,u}$) [25].

We show that the common classes of quantitative automata always express sup-closed properties, which will simplify the study of their safety and liveness.

▶ **Proposition 2.2.** *Let* Val $\in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$*. Every* Val*-automaton expresses a property that is* sup*-closed. Furthermore its top value is rational, attainable by a run, and can be computed in* PTIME*.*

3 Subroutine: Constant-Function Check

We will show that the problems of whether a given automaton is safe or live are closely related to the problem of whether an automaton expresses a constant function, motivating its study in this section. We first prove the problem hardness by reduction from the universality of nondeterministic finite-state automata (NFAs) and reachability automata.

▶ **Lemma 3.1.** *Let* Val $\in \{\mathsf{Sup}, \mathsf{Inf}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$*. It is* PSPACE*-hard to decide whether a* Val*-automaton* $\mathcal{A}$ *expresses a constant function.*

A simple solution to the problem is to check whether the given automaton $\mathcal{A}$ is equivalent to an automaton $\mathcal{B}$ expressing the constant top value of $\mathcal{A}$, which is computable in PTIME by Proposition 2.2. For some automata classes, it is good enough for a matching upper bound.

▶ **Proposition 3.2.** *Deciding whether an* Inf-*,* Sup-*,* LimInf-*, or* LimSup-*automaton expresses a constant function is* PSPACE*-complete.*

Yet, this simple approach does not work for DSum-automata, whose equivalence is an open problem, and for limit-average automata, whose equivalence is undecidable [15].

For DSum-automata, our alternative solution removes "non-optimal" transitions from the automaton and then reduces the problem to the universality problem of NFAs.

▶ **Theorem 3.3.** *Deciding whether a* DSum*-automaton expresses a constant function is* PSPACE*-complete.*

The solution for limit-average automata is more involved. It is based on a reduction to the limitedness problem of distance automata, which is known to be in PSPACE [21, 36, 22, 31]. We start with presenting Johnson's algorithm, which we will use for manipulating the transition weights of the given automaton, and proving some properties of distance automata, which we will need for the reduction.

A *weighted graph* is a directed graph $G = \langle V, E \rangle$ equipped with a weight function $\gamma : E \to \mathbb{Z}$. The cost of a path $p = v_0, v_1, \ldots, v_k$ is $\gamma(p) = \sum_{i=0}^{k-1} \gamma(v_i, v_{i+1})$.

▶ **Proposition 3.4** (Johnson's Algorithm [28, Lem. 2 and Thms. 4 and 5])**.** *Consider a weighted graph* $G = \langle V, E \rangle$ *with weight function* $\gamma : E \to \mathbb{Z}$*, such that* G *has no negative cycles according to* γ*. We can compute in* PTIME *functions* $h : V \to \mathbb{Z}$ *and* $\gamma' : E \to \mathbb{N}$ *such that for every path* $p = v_0, v_1, \ldots, v_k$ *in* G *it holds that* $\gamma'(p) = \gamma(p) + h(v_0) - h(v_k)$*.*

▶ Remark. Proposition 3.4 is stated for graphs, while we will apply it for graphs underlying automata, which are multi-graphs, namely having several transitions between the same pairs of states. Nevertheless, to see that Johnson's algorithm holds also in our case, one can change every automaton to an equivalent one whose underlying graph is a standard graph, by splitting every state into several states, each having a single incoming transition.

A *distance automaton* is a weighted automaton over the tropical semiring (a.k.a., min-plus semiring) with weights in $\{0,1\}$. It can be viewed as a quantitative automaton over finite words with transition weights in $\{0,1\}$ and the value function of summation, extended with accepting states. A distance automaton is of *limited distance* if there exists a bound on the automaton's values on all accepted words.

Lifting limitedness to infinite words, we have by König's lemma that a total distance automaton of limited distance b, in which all states are accepting, is also guaranteed to have a run whose weight summation is bounded by b on every infinite word.

▶ **Proposition 3.5.** *Consider a total distance automaton $\mathcal{D}$ of limited distance b, in which all states are accepting. Then for every infinite word w, there exists an infinite run of $\mathcal{D}$ on w whose summation of weights (considering only the transition weights and ignoring the final weights of states), is bounded by b.*

Lifting nonlimitedness to infinite words, it may not suffice for our purposes to have an infinite word on which all runs of the distance automaton are unbounded, as their limit-average value might still be 0. Yet, thanks to the following lemma, we are able to construct an infinite word on which the limit-average value is strictly positive.

▶ **Lemma 3.6.** *Consider a total distance automaton $\mathcal{D}$ of unlimited distance, in which all states are accepting. Then there exists a finite nonempty word u, such that $\mathcal{D}(u) = 1$ and the possible runs of $\mathcal{D}$ on u lead to a set of states U, such that the distance automaton that is the same as $\mathcal{D}$ but with U as the set of its initial states is also of unlimited distance.*

Using Propositions 3.4 and 3.5 and Lemma 3.6 we are in position to solve our problem by reduction to the limitedness problem of distance automata.

▶ **Theorem 3.7.** *Deciding whether a* LimInfAvg- *or* LimSupAvg-*automaton expresses a constant function, for a given constant or any constant, is PSPACE-complete.*

4 Quantitative Safety

The membership problem for quantitative properties asks, given a property $\Phi : \Sigma^\omega \to \mathbb{D}$, a word $w \in \Sigma^\omega$, and a value $v \in \mathbb{D}$, whether $\Phi(w) \geq v$ holds [10]. Safety of quantitative properties is defined from the perspective of membership queries [25]. Intuitively, a property is safe when each wrong membership hypothesis has a finite prefix to witness the violation. The safety closure of a given property maps each word to the greatest lower bound over its prefixes of the least upper bound of possible values.

▶ **Definition 4.1** (Safety [25]). *A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* safe *when for every $w \in \Sigma^\omega$ and value $v \in \mathbb{D}$ with $\Phi(w) \not\geq v$, there is a prefix $u \prec w$ such that $\sup_{w' \in \Sigma^\omega} \Phi(uw') \not\geq v$. The* safety closure *of a property Φ is the property defined by $SafetyCl(\Phi)(w) = \inf_{u \prec w} \sup_{w' \in \Sigma^\omega} \Phi(uw')$ for all $w \in \Sigma^\omega$.*

We remark that (i) a property is safe iff it defines the same function as its safety closure [25, Thm. 9], and (ii) the safety closure of a property is the least safety property that bounds the given property from above [25, Prop. 6]. Co-safety of quantitative properties and the co-safety closure is defined symmetrically.

▶ **Definition 4.2** (Co-safety [25]). *A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* co-safe *when for every $w \in \Sigma^\omega$ and value $v \in \mathbb{D}$ with $\Phi(w) \not\leq v$, there exists a prefix $u \prec w$ such that $\inf_{w' \in \Sigma^\omega} \Phi(uw') \not\leq v$. The* co-safety closure *of a property Φ is the property defined by $\mathit{CoSafetyCl}(\Phi)(w) = \sup_{u \prec w} \inf_{w' \in \Sigma^\omega} \Phi(uw')$ for all $w \in \Sigma^\omega$.*

Consider the case of a server that processes incoming requests and approves them accordingly. The quantitative property for the minimal response time of such a server is safe, while its maximal response time is co-safe [25, Examples 3 and 26]. Although these are sup- and inf-closed properties, safety and co-safety are independent of sup- and inf-closedness. To witness, consider the alphabet $\Sigma = \{a, b\}$ and the value domain $\mathbb{D} = \{x, y, \bot, \top\}$ where x and y are incomparable, and define $\Phi(w) = x$ if $a \prec w$ and $\Phi(w) = y$ if $b \prec w$.

▶ **Proposition 4.3.** *There is a property Φ that is safe and co-safe but neither* sup- *nor* inf*-closed.*

The Cantor space of infinite words is the set Σ^ω with the metric $\mu : \Sigma^\omega \times \Sigma^\omega \to [0, 1]$ such that $\mu(w, w) = 0$ and $\mu(w, w') = 2^{-|u|}$ where $u \in \Sigma^*$ is the longest common prefix of $w, w' \in \Sigma^\omega$ with $w \neq w'$. Given a boolean property $P \subseteq \Sigma^\omega$, the topological closure $\mathit{TopolCl}(P)$ of P is the smallest closed set (i.e., boolean safety property) that contains P, and the topological interior $\mathit{TopolInt}(P)$ of P is the greatest open set (i.e., boolean co-safety property) that is contained in P.

We show the connection between the quantitative safety (resp. co-safety) closure and the topological closure (resp. interior) through sup-closedness (resp. inf-closedness). Note that the sup-closedness assumption makes the quantitative safety closure values realizable. This guarantees that for every value v, every word whose safety closure value is at least v belongs to the topological closure of the set of words whose property values are at least v.

▶ **Theorem 4.4.** *Consider a property $\Phi : \Sigma^\omega \to \mathbb{D}$ and a threshold $v \in \mathbb{D}$. If Φ is* sup*-closed, then $(\mathit{SafetyCl}(\Phi))_{\geq v} = \mathit{TopolCl}(\Phi_{\geq v})$. If Φ is* inf*-closed, then $(\mathit{CoSafetyCl}(\Phi))_{\leq v} = \mathit{TopolInt}(\Phi_{\leq v})$.*

For studying the safety of automata, we first provide alternative characterizations of quantitative safety through threshold safety, which bridges the gap between the boolean and the quantitative settings, and continuity of functions. These hold for all properties on totally-ordered value domains, and in particular for those expressed by quantitative automata. Then, we extend the safety notions from properties to value functions, allowing us to characterize families of safe quantitative automata. Finally, we provide algorithms to construct the safety closure of a given automaton $\mathcal{A}$ and to decide whether $\mathcal{A}$ is safe.

4.1 Threshold Safety and Continuity

In this section, we define threshold safety to connect the boolean and the quantitative settings. It turns out that quantitative safety and threshold safety coincide on totally-ordered value domains. Furthermore, these value domains enable a purely topological characterization of quantitative safety properties in terms of their continuity.

▶ **Definition 4.5** (Threshold safety). *A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* threshold safe *when for every $v \in \mathbb{D}$ the boolean property $\Phi_{\geq v} = \{w \in \Sigma^\omega \mid \Phi(w) \geq v\}$ is safe (and thus $\Phi_{\not\geq v}$ is co-safe). Equivalently, for every $w \in \Sigma^\omega$ and $v \in \mathbb{D}$ if $\Phi(w) \not\geq v$ then there exists $u \prec w$ such that for all $w' \in \Sigma^\omega$ we have $\Phi(uw') \not\geq v$.*

▶ **Definition 4.6** (Threshold co-safety). *A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* threshold co-safe *when for every $v \in \mathbb{D}$ the boolean property $\Phi_{\not\leq v} = \{w \in \Sigma^\omega \mid \Phi(w) \not\leq v\}$ is co-safe (and thus $\Phi_{\leq v}$ is safe). Equivalently, for every $w \in \Sigma^\omega$ and $v \in \mathbb{D}$ if $\Phi(w) \not\leq v$ then there exists $u \prec w$ such that for all $w' \in \Sigma^\omega$ we have $\Phi(uw') \not\leq v$.*

In general, quantitative safety implies threshold safety, but the converse need not hold with respect to partially-ordered value domains. To witness, consider the value domain $\mathbb{D} = [0,1] \cup \{x\}$ where x is such that $0 < x$ and $x < 1$, but it is incomparable with all $v \in (0,1)$, while within $[0,1]$ there is the standard order. Let Φ be a property defined over $\Sigma = \{a,b\}$ as follows: $\Phi(w) = x$ if $w = a^\omega$, $\Phi(w) = 2^{-|w|_a}$ if $w \in \Sigma^* b^\omega$, and $\Phi(w) = 0$ otherwise. We show that Φ is threshold safe but not safe.

▶ **Proposition 4.7.** *Every safety property is threshold safe, but there is a threshold-safety property that is not safe.*

While for a fixed threshold, safety and threshold safety do not necessarily overlap even on totally-ordered domains, once quantifying over all thresholds, they do.

▶ **Theorem 4.8.** *Let $\mathbb{D}$ be a totally-ordered value domain. A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is safe iff it is threshold safe.*

We move next to the relation between safety and continuity. We recall some standard definitions; more about it can be found in textbooks, e.g., [19, 27]. A *topology* of a set X can be defined to be its collection τ of open subsets, and the pair (X, τ) stands for a *topological space*. It is *metrizable* when there exists a distance function (metric) d on X such that the topology induced by d on X is τ.

Recall that we take Σ^ω as a Cantor space with the metric μ defined as in Section 4. Consider a totally-ordered value domain $\mathbb{D}$. For each element $v \in \mathbb{D}$, let $L_v = \{v' \in \mathbb{D} \mid v' < v\}$ and $R_v = \{v' \in \mathbb{D} \mid v < v'\}$. The *order topology* on $\mathbb{D}$ is generated by the set $\{L_v \mid v \in \mathbb{D}\} \cup \{R_v \mid v \in \mathbb{D}\}$. Moreover, the *left order topology* (resp. *right order topology*) is generated by the set $\{L_v \mid v \in \mathbb{D}\}$ (resp. $\{R_v \mid v \in \mathbb{D}\}$). For a given property $\Phi : \Sigma^\omega \to \mathbb{D}$ and a set $V \subseteq \mathbb{D}$ of values, the *preimage* of V on Φ is defined as $\Phi^{-1}(V) = \{w \in \Sigma^\omega \mid \Phi(w) \in V\}$.

A property $\Phi : \Sigma^\omega \to \mathbb{D}$ on a topological space $\mathbb{D}$ is *continuous* when for every open subset $V \subseteq \mathbb{D}$ the preimage $\Phi^{-1}(V) \subseteq \Sigma^\omega$ is open. In [25, 26], a property Φ is defined as upper semicontinuous when $\Phi(w) = \lim_{u \prec w} \sup_{w' \in \Sigma^\omega} \Phi(uw')$, extending the standard definition for functions on extended reals to functions from infinite words to complete lattices. This characterizes safety properties since it is an equivalent condition to a property defining the same function as its safety closure [25, Thm. 9]. We complete the picture by providing a purely topological characterization of safety properties in terms of their continuity in totally-ordered value domains.

▶ **Theorem 4.9.** *Let $\mathbb{D}$ be a totally-ordered value domain. A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is safe (resp. co-safe) iff it is continuous with respect to the left (resp. right) order topology on $\mathbb{D}$.*

Observe that a property is continuous with respect to the order topology on $\mathbb{D}$ iff it is continuous with respect to both left and right order topologies on $\mathbb{D}$. Then, we immediately obtain the following.

▶ **Corollary 4.10.** *Let $\mathbb{D}$ be a totally-ordered value domain. A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is safe and co-safe iff it is continuous with respect to the order topology on $\mathbb{D}$.*

Now, we shift our focus to totally-ordered value domains whose order topology is metrizable. We provide a general definition of discounting properties on such domains.

▶ **Definition 4.11** (Discounting). *Let $\mathbb{D}$ be a totally-ordered value domain for which the order topology is metrizable with a metric d. A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* discounting *when for every $\varepsilon > 0$ there exists $n \in \mathbb{N}$ such that for every $u \in \Sigma^n$ and $w, w' \in \Sigma^\omega$ we have $d(\Phi(uw), \Phi(uw')) < \varepsilon$.*

Intuitively, a property is discounting when the range of potential values for every word converges to a singleton. As an example, consider the following discounted safety property: Given a boolean safety property P, let Φ be a quantitative property such that $\Phi(w) = 1$ if $w \in P$, and $\Phi(w) = 2^{-|u|}$ if $w \notin P$, where $u \prec w$ is the shortest bad prefix of w for P. We remark that our definition captures the previous definitions of discounting given in [14, 1].

▶ Remark. Notice that the definition of discounting coincides with uniform continuity. Since Σ^ω equipped with Cantor distance is a compact space [16], every continuous property is also uniformly continuous by Heine-Cantor theorem, and thus discounting.

As an immediate consequence, we obtain the following.

▶ **Corollary 4.12.** *Let $\mathbb{D}$ be a totally-ordered value domain for which the order topology is metrizable. A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is safe and co-safe iff it is discounting.*

4.2 Safety of Value Functions

In this section, we focus on the value functions of quantitative automata, which operate on the value domain of real numbers. In particular, we carry the definitions of safety, co-safety, and discounting to value functions. This allows us to characterize safe (resp. co-safe, discounting) value functions as those for which all automata with this value function are safe (resp. co-safe, discounting). Moreover, we characterize discounting value functions as those that are safe and co-safe.

Recall that we consider the value functions of quantitative automata to be bounded from below and above for every finite input domain $V \subset \mathbb{Q}$. As the set V^ω can be taken as a Cantor space, just like Σ^ω, we can carry the notions of safety, co-safety, and discounting from properties to value functions.

▶ **Definition 4.13** (Safety and co-safety of value functions). *A value function $\mathsf{Val} : \mathbb{Q}^\omega \to \mathbb{R}$ is* safe *when for every finite subset $V \subset \mathbb{Q}$, infinite sequence $x \in V^\omega$, and value $v \in \mathbb{R}$, if $\mathsf{Val}(x) < v$ then there exists a finite prefix $z \prec x$ such that $\sup_{y \in V^\omega} \mathsf{Val}(zy) < v$. Similarly, a value function $\mathsf{Val} : \mathbb{Q}^\omega \to \mathbb{R}$ is* co-safe *when for every finite subset $V \subset \mathbb{Q}$, infinite sequence $x \in V^\omega$, and value $v \in \mathbb{R}$, if $\mathsf{Val}(x) > v$ then there exists a finite prefix $z \prec x$ such that $\inf_{y \in V^\omega} \mathsf{Val}(zy) > v$.*

▶ **Definition 4.14** (Discounting value function). *A value function $\mathsf{Val} : \mathbb{Q}^\omega \to \mathbb{R}$ is* discounting *when for every finite subset $V \subset \mathbb{Q}$ and every $\varepsilon > 0$ there exists $n \in \mathbb{N}$ such that for every $x \in V^n$ and $y, y' \in V^\omega$ we have $|\mathsf{Val}(xy) - \mathsf{Val}(xy')| < \varepsilon$.*

We remark that by [25, Thms. 20 and 27], the value function Inf is safe and Sup is co-safe; moreover, the value function DSum is discounting by definition. Now, we characterize the safety (resp. co-safety) of a given value function by the safety (resp. co-safety) of the automata family it defines. We emphasize that the proofs of the two statement are not dual. In particular, exhibiting a finite set of weights that falsifies the safety of a value function from a nonsafe automaton requires a compactness argument.

▶ **Theorem 4.15.** *Consider a value function* Val*. All* Val*-automata are safe (resp. co-safe) iff* Val *is safe (resp. co-safe).*

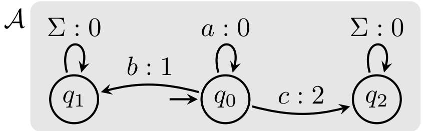


Figure 2 A Sup-automaton whose safety closure cannot be expressed by a Sup-automaton.

Thanks to the remark following Definition 4.11, for any finite set of weights $V \subset \mathbb{Q}$, a value function is discounting iff it is continuous on the Cantor space V^ω. We leverage this observation to characterize discounting value functions as those that are both safe and co-safe.

▶ **Theorem 4.16.** *A value function is discounting iff it is safe and co-safe.*

As a consequence of Corollary 4.12 and Theorems 4.15–4.16, we obtain the following.

▶ **Corollary 4.17.** *All* Val*-automata are discounting iff* Val *is discounting.*

4.3 Safety of Quantitative Automata

We now switch our focus from generic value functions to families of quantitative automata defined by the common value functions Inf, Sup, LimInf, LimSup, LimInfAvg, LimSupAvg, and DSum. As remarked in Section 4.2, the value functions Inf and DSum are safe, thus all Inf-automata and DSum-automata express a safety property by Theorem 4.15. Below, we focus on the remaining value functions of interest.

Given a Val-automaton $\mathcal{A}$ where Val is one of the nonsafe value functions above, we describe (i) a construction of an automaton that expresses the safety closure of $\mathcal{A}$, and (ii) an algorithm to decide whether $\mathcal{A}$ is safe.

For these value functions, we can construct the safety closure as an Inf-automaton.

▶ **Theorem 4.18.** *Let* Val $\in$ {Sup, LimInf, LimSup, LimInfAvg, LimSupAvg}*. Given a* Val*-automaton $\mathcal{A}$, we can construct in* PTime *an* Inf*-automaton that expresses its safety closure.*

For the prefix-independent value functions we study, the safety-closure automaton we construct in Theorem 4.18 can be taken as a deterministic automaton with the same value function.

▶ **Theorem 4.19.** *Let* Val $\in$ {LimInf, LimSup, LimInfAvg, LimSupAvg}*. Given a* Val*-automaton $\mathcal{A}$, we can construct in* PTime *a* Val*-automaton that expresses its safety closure and can be determinized in* ExpTime*.*

In contrast, this is not possible in general for Sup-automata, as Figure 2 witnesses.

▶ **Proposition 4.20.** *Some* Sup*-automaton admits no* Sup*-automata that expresses its safety closure.*

We first prove the problem hardness by reduction from constant-function checks.

▶ **Lemma 4.21.** *Let* Val $\in$ {Sup, LimInf, LimSup, LimInfAvg, LimSupAvg}*. It is* PSpace*-hard to decide whether a* Val*-automaton is safe.*

For automata classes with PSpace equivalence check, a matching upper bound is straightforward by comparing the given automaton and its safety-closure automaton.

▶ **Theorem 4.22.** *Deciding whether a* Sup-, LimInf-, *or* LimSup-*automaton expresses a safety property is* PSPACE*-complete.*

On the other hand, even though equivalence of limit-average automata is undecidable [15], we are able to provide a decision procedure using as a subroutine our algorithm to check whether a given limit-average automaton expresses a constant function (see Theorem 3.7). The key idea is to construct a limit-average automaton that expresses the constant function 0 iff the original automaton is safe. Our approach involves the determinization of the safety-closure automaton, resulting in an ExpSpace complexity.

▶ **Theorem 4.23.** *Deciding whether a* LimInfAvg- *or* LimSupAvg-*automaton expresses a safety property is in* ExpSpace.

5 Quantitative Liveness

The definition of quantitative liveness, similarly to that of quantitative safety, comes from the perspective of the quantitative membership problem [25]. Intuitively, a property is live when for every word whose value is less than the top, there is a wrong membership hypothesis without a finite prefix to witness the violation.

▶ **Definition 5.1** (Liveness and co-liveness [25]). *A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* live *when for all $w \in \Sigma^\omega$, if $\Phi(w) < \top$, then there exists a value $v \in \mathbb{D}$ such that $\Phi(w) \not\geq v$ and for all prefixes $u \prec w$, we have $\sup_{w' \in \Sigma^\omega} \Phi(uw') \geq v$. Similarly, a property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* co-live *when for all $w \in \Sigma^\omega$, if $\Phi(w) > \bot$, then there exists a value $v \in \mathbb{D}$ such that $\Phi(w) \not\leq v$ and for all prefixes $u \prec w$, we have $\inf_{w' \in \Sigma^\omega} \Phi(uw') \leq v$.*

As an example, consider a server that receives requests and issues grants. The server's maximum response time is live, while its minimum response time is co-live, and its average response time is both live and co-live [25, Examples 41 and 42].

First, we provide alternative characterizations of quantitative liveness for sup-closed properties by threshold liveness, which bridges the gap between the boolean and the quantitative settings, and top liveness. Then, we provide algorithms to check liveness of quantitative automata, and to decompose them into a safety automaton and a liveness automaton.

5.1 Threshold Liveness and Top Liveness

Threshold liveness connects a quantitative property and the boolean liveness of the sets of words whose values exceed a threshold value.

▶ **Definition 5.2** (Threshold liveness and co-liveness). *A property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* threshold live *when for every $v \in \mathbb{D}$ the boolean property $\Phi_{\geq v} = \{w \in \Sigma^\omega \mid \Phi(w) \geq v\}$ is live (and thus $\Phi_{\not\geq v}$ is co-live). Equivalently, Φ is threshold live when for every $u \in \Sigma^*$ and $v \in \mathbb{D}$ there exists $w \in \Sigma^\omega$ such that $\Phi(uw) \geq v$. Similarly, a property $\Phi : \Sigma^\omega \to \mathbb{D}$ is* threshold co-live *when for every $v \in \mathbb{D}$ the boolean property $\Phi_{\not\leq v} = \{w \in \Sigma^\omega \mid \Phi(w) \not\leq v\}$ is co-live (and thus $\Phi_{\leq v}$ is live). Equivalently, Φ is threshold co-live when for every $u \in \Sigma^*$ and $v \in \mathbb{D}$ there exists $w \in \Sigma^\omega$ such that $\Phi(uw) \leq v$.*

A set $P \subseteq \Sigma^\omega$ is dense in Σ^ω when its topological closure equals Σ^ω, i.e., $\mathit{TopolCl}(P) = \Sigma^\omega$. We relate threshold liveness with the topological denseness of a single set of words.

▶ **Proposition 5.3.** *A property Φ is threshold live iff the set $\{w \in \Sigma^\omega \mid \Phi(w) = \top\}$ is dense in Σ^ω.*

Liveness is characterized by the safety closure being strictly greater than the property whenever possible [25, Thm. 37]. Top liveness puts an additional requirement on liveness: the safety closure of the property should not only be greater than the original property but also equal to the top value.

▶ **Definition 5.4** (Top liveness and bottom co-liveness). *A property Φ is* top live *when $SafetyCl(\Phi)(w) = \top$ for every $w \in \Sigma^\omega$. Similarly, a property Φ is* bottom co-live *when $CoSafetyCl(\Phi)(w) = \bot$ for every $w \in \Sigma^\omega$.*

We provide a strict hierarchy of threshold-liveness, top-liveness, and liveness.

▶ **Proposition 5.5.** *Every threshold-live property is top live, but not vice versa; and every top-live property is live, but not vice versa.*

Top liveness does not imply threshold liveness, but it does imply a weaker form of it.

▶ **Proposition 5.6.** *For every top-live property Φ and value $v < \top$, the set $\Phi_{\geq v}$ is live in the boolean sense.*

While the three liveness notions differ in general, they do coincide for sup-closed properties.

▶ **Theorem 5.7.** *A* sup*-closed property is live iff it is top live iff it is threshold live.*

5.2 Liveness of Quantitative Automata

We start with the problem of checking whether a quantitative automaton is live, and continue with quantitative safety-liveness decomposition.

We first provide a hardness result by reduction from constant-function checks.

▶ **Lemma 5.8.** *Let* $\mathsf{Val} \in \{\mathsf{Inf}, \mathsf{Sup}, \mathsf{LimInf}, \mathsf{LimSup}, \mathsf{LimInfAvg}, \mathsf{LimSupAvg}, \mathsf{DSum}\}$*. It is PSPACE-hard to decide whether a* Val*-automaton $\mathcal{A}$ is live.*

For automata classes whose safety closure can be expressed as Inf-automata, we provide a matching upper bound by simply checking the universality of the safety closure with respect to its top value. For DSum-automata, whose universality problem is open, our solution is based on our constant-function-check algorithm (see Theorem 3.3).

▶ **Theorem 5.9.** *Deciding whether an* Inf-*,* Sup-*,* LimInf-*,* LimSup-*,* LimInfAvg-*,* LimSupAvg- *or* DSum-*automaton expresses a liveness property is PSPACE-complete.*

We turn to safety-liveness decomposition, and start with the simple case of Inf- and DSum-automata, which are guaranteed to be safe. Their decomposition thus consists of only generating a liveness component, which can simply express a constant function that is at least as high as the maximal possible value of the original automaton $\mathcal{A}$. Assuming that the maximal transition weight of $\mathcal{A}$ is fixed, it can be done in constant time.

Considering Sup-automata, recall that their safety closure might not be expressible by Sup-automata (Proposition 4.20). Therefore, our decomposition of deterministic Sup-automata takes the safety component as an Inf-automaton. The key idea is to copy the state space of the original automaton and manipulate the transition weights depending on how they compare with the safety-closure automaton.

▶ **Theorem 5.10.** *Given a deterministic* Sup*-automaton $\mathcal{A}$, we can construct in PTIME a deterministic safety* Inf*-automaton $\mathcal{B}$ and a deterministic liveness* Sup*-automaton $\mathcal{C}$, such that $\mathcal{A}(w) = \min(\mathcal{B}(w), \mathcal{C}(w))$ for every infinite word w.*

Using the same idea, but with a slightly more involved reasoning, we show a safety-liveness decomposition for deterministic LimInf- and LimSup-automata.

▶ **Theorem 5.11.** *Let* $\mathsf{Val} \in \{\mathsf{LimInf}, \mathsf{LimSup}\}$*. Given a deterministic* Val*-automaton* $\mathcal{A}$*, we can construct in* PTIME *a deterministic safety* Val*-automaton* $\mathcal{B}$ *and a deterministic liveness* Val*-automaton* $\mathcal{C}$*, such that* $\mathcal{A}(w) = \min(\mathcal{B}(w), \mathcal{C}(w))$ *for every infinite word* w*.*

Considering nondeterministic Sup- and LimInf-automata, they can be decomposed by first determinizing them at an exponential cost [10, Thm. 14]. For nondeterministic LimSup-automata, which cannot always be determinized, we leave the problem open. We also leave open the question of whether $\mathsf{LimInfAvg}$- and $\mathsf{LimSupAvg}$-automata are closed under safety-liveness decomposition.

6 Conclusions

We studied, for the first time, the quantitative safety-liveness dichotomy for properties expressed by Inf, Sup, LimInf, LimSup, $\mathsf{LimInfAvg}$, $\mathsf{LimSupAvg}$, and DSum automata. To this end, we characterized the quantitative safety and liveness of automata by their boolean counterparts, connected them to topological continuity and denseness, and solved the constant-function problem for these classes of automata. We presented automata-theoretic constructions for the safety closure of these automata and decision procedures for checking their safety and liveness. We proved that the value function Inf yields a class of safe automata and DSum both safe and co-safe. For some automata classes, we provided a decomposition of an automaton into a safe and a live component. We emphasize that the safety component of our decomposition algorithm is the safety closure, and thus the best safe approximation of a given automaton.

We focused on quantitative automata [10] because their totally-ordered value domain and their sup-closedness make quantitative safety and liveness behave in particularly natural ways; a corresponding investigation of weighted automata [35] remains to be done. We left open the problems of the safety-liveness decomposition of limit-average automata, the complexity gap in the safety check of limit-average automata, and the study of co-safety and co-liveness for nondeterministic quantitative automata, which is not symmetric to safety and liveness due to the nonsymmetry in resolving nondeterminism by the supremum value of all possible runs.

References

1 Shaull Almagor, Udi Boker, and Orna Kupferman. Discounting in LTL. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2014. doi:10.1007/978-3-642-54862-8_37.

2 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.

3 Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Comput.*, 2(3):117–126, 1987. doi:10.1007/BF01782772.

4 Suguman Bansal and Moshe Y. Vardi. Safety and co-safety comparator automata for discounted-sum inclusion. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification – 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 60–78. Springer, 2019. doi:10.1007/978-3-030-25540-4_4.

5 Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 – July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009. doi:10.1007/978-3-642-02658-4_14.

6 Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. Graph games and reactive synthesis. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 921–962. Springer, 2018. doi:10.1007/978-3-319-10575-8_27.

7 Udi Boker. Quantitative vs. weighted automata. In *Proc. of Reachbility Problems*, pages 1–16, 2021.

8 Udi Boker and Karoliina Lehtinen. Token games and history-deterministic quantitative automata. In Patricia Bouyer and Lutz Schröder, editors, *Foundations of Software Science and Computation Structures – 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13242 of *Lecture Notes in Computer Science*, pages 120–139. Springer, 2022. doi:10.1007/978-3-030-99253-8_7.

9 Patricia Bouyer, Nicolas Markey, Mickael Randour, Kim G. Larsen, and Simon Laursen. Average-energy games. *Acta Informatica*, 55(2):91–127, 2018. doi:10.1007/s00236-016-0274-1.

10 Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. *ACM Trans. Comput. Log.*, 11(4):23:1–23:38, 2010. doi:10.1145/1805950.1805953.

11 Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop. Quantitative monitor automata. In Xavier Rival, editor, *Static Analysis – 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 23–38. Springer, 2016. doi:10.1007/978-3-662-53413-7_2.

12 Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification – 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 383–401. Springer, 2016. doi:10.1007/978-3-319-41540-6_21.

13 Luca de Alfaro, Marco Faella, and Mariëlle Stoelinga. Linear and branching metrics for quantitative transition systems. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004. doi:10.1007/978-3-540-27836-8_11.

14 Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Discounting the future in systems theory. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 – July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 1022–1037. Springer, 2003. doi:10.1007/3-540-45061-0_79.

15 Aldric Degorre, Laurent Doyen, Raffaella Gentilini, Jean-François Raskin, and Szymon Torunczyk. Energy and mean-payoff games with imperfect information. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2010. doi:10.1007/978-3-642-15205-4_22.

16 Volker Diekert and Martin Leucker. Topology, monitorable properties and runtime verification. *Theor. Comput. Sci.*, 537:29–41, 2014. doi:10.1016/j.tcs.2014.02.052.

17 Uli Fahrenberg. A generic approach to quantitative verification. *CoRR*, abs/2204.11302, 2022. doi:10.48550/arXiv.2204.11302.

18 Thomas Ferrère, Thomas A. Henzinger, and N. Ege Saraç. A theory of register monitors. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 394–403. ACM, 2018. doi:10.1145/3209108.3209194.

19 Theodore W Gamelin and Robert Everist Greene. *Introduction to topology*. Courier Corporation, 1999.

20 Felipe Gorostiaga and César Sánchez. Monitorability of expressive verdicts. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods – 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 693–712. Springer, 2022. doi:10.1007/978-3-031-06773-0_37.

21 K. Hashiguchi. Limitedness theorem on finite automata with distance functions. *Journal of computer and system sciences*, 24(2):233–244, 1982.

22 K. Hashiguchi. New upper bounds to the limitedness of distance automata. *Theoretical Computer Science*, 233(1-2):19–32, 2000.

23 Thomas A. Henzinger. Quantitative reactive modeling and verification. *Comput. Sci. Res. Dev.*, 28(4):331–344, 2013. doi:10.1007/s00450-013-0251-7.

24 Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. Abstract monitors for quantitative specifications. In Thao Dang and Volker Stolz, editors, *Runtime Verification – 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume 13498 of *Lecture Notes in Computer Science*, pages 200–220. Springer, 2022. doi:10.1007/978-3-031-17196-3_11.

25 Thomas A. Henzinger, Nicolas Mazzocchi, and N. Ege Saraç. Quantitative safety and liveness. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures – 26th International Conference, FoSSaCS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, volume 13992 of *Lecture Notes in Computer Science*, pages 349–370. Springer, 2023. doi:10.1007/978-3-031-30829-1_17.

26 Thomas A. Henzinger and N. Ege Saraç. Quantitative and approximate monitoring. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 – July 2, 2021*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470547.

27 Hendrik Jan Hoogeboom and Grzegorz Rozenberg. Infinitary languages: Basic theory an applications to concurrent systems. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Current Trends in Concurrency, Overviews and Tutorials*, volume 224 of *Lecture Notes in Computer Science*, pages 266–342. Springer, 1986. doi:10.1007/BFb0027043.

28 D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

29 Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods Syst. Des.*, 19(3):291–314, 2001. doi:10.1023/A:1011254632723.

30 Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.

31 H. Leung and V. Podolskiy. The limitedness problem on distance automata: Hashiguchi's method revisited. *Theoretical Computer Science*, 310(1-3):147–158, 2004.

32 Yongming Li, Manfred Droste, and Lihui Lei. Model checking of linear-time properties in multi-valued systems. *Inf. Sci.*, 377:51–74, 2017. doi:10.1016/j.ins.2016.10.030.

33 Dejan Nickovic, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. *Int. J. Softw. Tools Technol. Transf.*, 22(6):741–758, 2020. doi:10.1007/s10009-020-00582-z.

34 Doron Peled and Klaus Havelund. Refining the safety-liveness classification of temporal properties according to monitorability. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? – Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2018. doi:10.1007/978-3-030-22348-9_14.

35 Marcel Paul Schützenberger. On the definition of a family of automata. *Inf. Control.*, 4(2-3):245–270, 1961. doi:10.1016/S0019-9958(61)80020-X.

36 I. Simon. On semigroups of matrices over the tropical semiring. *RAIRO-Theoretical Informatics and Applications*, 28(3-4):277–294, 1994.

37 A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects Comput.*, 6(5):495–512, 1994. doi:10.1007/BF01211865.

38 Sigal Weiner, Matan Hasson, Orna Kupferman, Eyal Pery, and Zohar Shevach. Weighted safety. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis – 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2013. doi:10.1007/978-3-319-02444-8_11.

History-Deterministic Vector Addition Systems

Sougata Bose
University of Liverpool, UK

David Purser
University of Liverpool, UK

Patrick Totzke
University of Liverpool, UK

Abstract

We consider history-determinism, a restricted form of non-determinism, for Vector Addition Systems with States (VASS) when used as acceptors to recognise languages of finite words. History-determinism requires that the non-deterministic choices can be resolved on-the-fly; based on the past and without jeopardising acceptance of any possible continuation of the input word.

Our results show that the history-deterministic (HD) VASS sit strictly between deterministic and non-deterministic VASS regardless of the number of counters. We compare the relative expressiveness of HD systems, and closure-properties of the induced language classes, with coverability and reachability semantics, and with and without ε-labelled transitions.

Whereas in dimension 1, inclusion and regularity remain decidable, from dimension two onwards, HD-VASS with suitable resolver strategies, are essentially able to simulate 2-counter Minsky machines, leading to several undecidability results: It is undecidable whether a VASS is history-deterministic, or if a language equivalent history-deterministic VASS exists. Checking language inclusion between history-deterministic 2-VASS is also undecidable.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Vector Addition Systems, History-determinism, Good-for Games

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.18

Related Version *Full Version*: `https://arxiv.org/abs/2305.01981` [7]

Funding We gratefully acknowledge support from the EPSRC, project number `EP/V025848/1` and the Royal Society, `IES\R3\203109`.

Acknowledgements The authors thank Ismaël Jecker, Piotr Hofman and Filip Mazowiecki for fruitful discussions.

1 Introduction

Vector addition systems with states (VASSs) are an established model of concurrency with extensive applications in modelling and analysis of hardware, software, chemical, biological and business processes. They are non-deterministic finite automata equipped with a fixed number of integer counters that may be incremented or decremented when changing control state, as long as they remain non-negative.

We explore the notion of *history-determinism* for VASSs when used as acceptors to define languages of finite words. History-determinism is a restricted form of non-determinism. In a nutshell, a non-deterministic automaton is history-deterministic (HD) if there exists a *resolver*, which is a strategy to stepwise produce a run for any input word given one letter at a time, in such a way that if there exist some accepting run on the given word then the run produced by the resolver is also accepting.

The original motivation for HDness comes from formal verification: most modelling formalisms incorporate some form of non-determinism, e.g., to over-approximate deterministic algorithms, to state specifications concisely, or to model system behaviour due to

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 18; pp. 18:1–18:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

uncontrollable external environments. However, for non-deterministic models, many formal analysis techniques require costly determinisation steps that are often the main barrier to efficient procedures. History-deterministic automata provide a middle ground: they are typically more succinct, or even more expressive, than their deterministic counterparts while preserving some of their good algorithmic properties. They were also called "good-for-games" as they preserve the winner of games under composition and thus allow solving games without determinisation.

Any resolver must always choose language-maximal successors, that is, the language of the chosen successor must also include the language of any alternative successor. When considering languages of finite words, being able to continue making language-maximal choices is even a sufficient condition for being a resolver. Therefore, in this case, resolvers can be assumed to be (configuration) positional: they base their decisions only on the current configuration and not the full history leading to it. Perhaps surprisingly, resolvers for VASSs are not necessarily monotone with respect to counter values, and may require more than just comparing counters to integer thresholds (see full version [7]).

Related Work. VASSs, also known as Petri nets or partially blind counter automata, have been studied intensively since their inception in the 1960s. Early works focussed on modelling capabilities, relative expressiveness and closure properties of their recognised languages [15, 13, 37, 22] but the bulk of research on VASSs concerns decidability and complexity of decision problems [24, 29, 33, 25, 21, 23, 2, 28, 10]. In order to define languages with VASSs, different definitions distinguish between coverability and reachability acceptance conditions, and whether or not silent (ε) transitions are permitted. Checking language emptiness amounts to testing coverability or reachability, which are EXPSPACE [33, 29] and Ackermann-complete [10] respectively. Many other decision problems are undecidable, such as checking language inclusion, bisimulation and related equivalences [20] as well as checking (language) regularity [23]. Universality is undecidable for reachability acceptance [37] and decidable for coverability acceptance, via a well-quasi-order argument but with extremely high complexity (Hyper-Ackermannian in general [21] and still Ackermannian in dimension 1 [19]). These negative results by and large rely on the presence of non-deterministic choice, which motivates restricted forms of non-determinism such as bounded ambiguity (that allows for decidable inclusion [9]) or the notion of history-determinism studied here.

VASS recognisable languages over infinite words are significantly more complex than their finite-word cousins, both topologically and in terms of decision problems: already 1-VASS with (cover) Büchi acceptance can recognise Σ_1^1-complete languages [35, 12] and have an undecidable universality problem [1]. Again, the added complexity is due to non-determinism (languages of deterministic models are Borel, lower in the analytical hierarchy).

History-determinism was introduced independently, with slightly different definitions, by Henzinger and Piterman [17] for solving games without determinisation, by Colcombet [8] for cost-functions, and by Kupferman, Safra, and Vardi [26] for recognising derived tree languages of word automata. These different definitions all coincide for finite automata [3] but not necessarily for more general quantitative automata [4].

Until now, history-determinism has mainly been studied for finite-state systems. In this paper we continue a recent line of work [14, 27, 11, 16, 6, 32] that studies the notion for infinite-state models capable of recognising languages beyond (ω-)regular ones. For infinite-state systems, deterministic models are in general less expressive, not just less succinct, than their non-deterministic counterparts. In some cases they can be determinised, such is the case for quantitative automata [4] and timed automata with safety and reachability acceptance [16]. In contrast, for pushdown automata [14] and Parikh automata (VASS with $\mathbb{Z}$-valued counters; [11]), and timed automata with co-Büchi acceptance, allowing history-determinism strictly increases expressiveness (and adds more closure properties) compared to the deterministic variant. Whenever HD automata are strictly less expressive than fully non-deterministic ones, one can reasonably ask if there exists an equivalent HD automaton for a given non-deterministic one. This language HDness question is undecidable for pushdown and Parikh-automata [14, 11]. In fact, even checking if a given (pushdown or Parikh) automaton is itself HD is undecidable (for Parikh automata this follows for example by the undecidability of 2-dim. robot games [31]). On the other hand, checking HDness for timed automata is decidable [16] and various models of quantitative automata [5].

Most closely related to our work is that of Prakash and Thejaswini [32] who study history-deterministic one counter automata (OCA; PDA with unary stack alphabet) and nets (OCN; 1-dimensional VASSs) with state-based (coverability) acceptance. They show that checking automata HDness and inclusion are undecidable for OCA but remain decidable for OCNs. A useful consequence of their construction is that for any OCN one can construct a language equivalent deterministic OCA (with zero-test), albeit with a doubly exponential blow-up. They do not consider closure properties and leave open whether history-deterministic OCNs can be determinised, are equally expressive as fully non-deterministic OCNs, or fall strictly in between in expressiveness. Our work extends and generalises this paper in several directions.

Our Contributions. We study history-deterministic VASSs on finite words and without restricting the dimension. We consider coverability and reachability acceptance conditions, with and without silent (ε) transitions, and in all cases study the relative expressiveness, closure properties, and related decision problems.

We show that HD VASSs are more expressive than deterministic, but less expressive than non-deterministic ones. The same is true for languages recognised by VASSs of any fixed dimensions k, which answers the open question in [32] for $k = 1$. In particular, we provide examples of 1-dim. HD VASSs for which no equivalent deterministic ones exist in any dimension k, and also demonstrate that HD VASSs are strictly more expressive than finitely sequential ones (another restricted form of non-determinism).

We show that HD VASS languages are closed under inverse homomorphisms and intersections for both coverability and reachability semantics, although sometimes necessarily increasing the dimension. Coverability languages are closed under unions, whereas reachability languages are not. Neither are closed under other standard operations, including complementation, concatenation, homomorphisms, iteration and commutative closures.

We report that HDness is not sufficient for decidability of inclusion checking, even for 2-dimensional VASSs. A direct consequence is the undecidability of checking HDness of a given 2-VASS, contrasting decidability in dimension 1. Further, it is undecidable to check if a given VASS has a HD equivalent, and also if a given HD VASS recognises a regular language.

2 Definitions

Vector-Addition Systems and their recognised languages. A k-dimensional *vector-addition system* (k-VASS) is a non-deterministic finite automaton whose transitions manipulate k non-negative integer counters. It is given by $\mathcal{A} = (\Sigma, Q, \delta, s_0, F)$ consisting of a finite alphabet Σ; a finite set of control states Q; a transition relation $\delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times \mathbb{Z}^k \times Q$; an initial state s_0; a subset $F \subseteq Q$ of final states.

For a transition $t = (s, a, e, s') \in \delta$ we sometimes write $\mathit{label}(t) \overset{\mathit{def}}{=} a$ for the letter from $\Sigma \cup \{\varepsilon\}$ it reads and $\mathit{effect}(t) \overset{\mathit{def}}{=} e$ for its *effect* on the counters. $\|\delta\|$ denotes the largest absolute effect among all transitions on any counter.

A VASS naturally induces an infinite-state labelled transition system in which each *configuration* is a pair $(s, v) \in Q \times \mathbb{N}^k$ comprising a control state and a *non-negative* integer vector. Every transition $t = (s, a, e, s') \in \delta$ gives rise to steps $(s, v) \xrightarrow{t} (s', v')$ for all $v, v' \in \mathbb{N}$ with $v' = v + e$. We will call a path $\rho = (s_0, v_0) \xrightarrow{t_1} (s_1, v_1) \xrightarrow{t_1} \ldots \xrightarrow{t_k} (s_k, v_k)$ a *run* of the VASS and say it is a *cycle* if $s_0 = s_k$. Its *effect* is the sum of all transition effects $\mathit{effect}(\rho) \overset{\mathit{def}}{=} \sum_{i=1}^{k} \mathit{effect}(t_i)$. A run ρ as above *reads* the word $\mathit{label}(\rho) = \mathit{label}(t_1)\mathit{label}(t_2)\ldots\mathit{label}(t_k) \in \Sigma^*$. It is *accepting* if it ends in a final configuration.

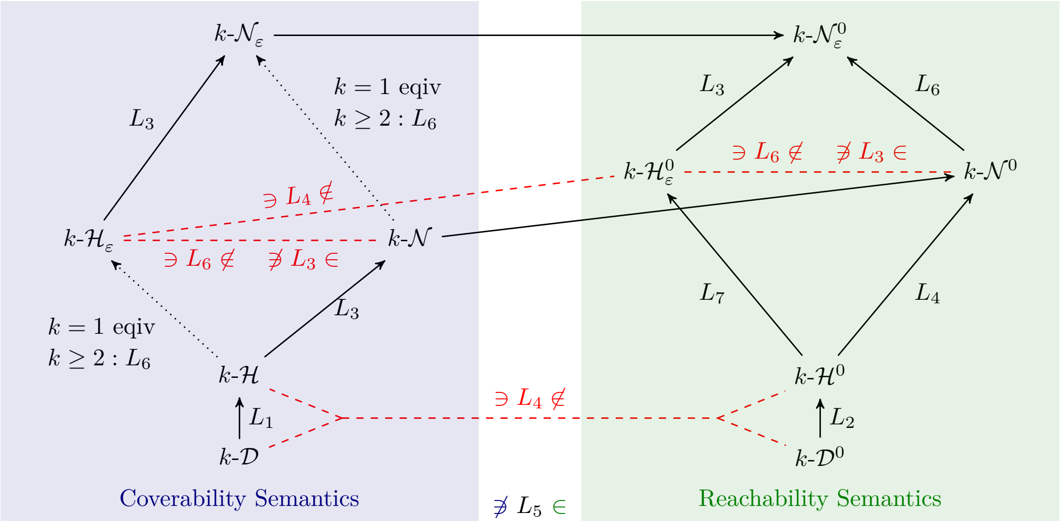
We consider two different definitions for what constitutes a final (also *accepting*) configurations: In the *coverability* semantics, the set of final configurations is $F \times \mathbb{N}$. In the *reachability* semantics, only configurations from $F \times \mathbf{0}$ are final. We define the *language* $\mathcal{L}_{\mathcal{A}}(s, v) \subseteq \Sigma^*$ of a configuration (s, v) to contain exactly all words read by some accepting run starting in (s, v) (we omit the subscript $\mathcal{A}$ if the VASS is clear from context). For notational convenience, we will lift this to sets $S \subseteq Q \times \mathbb{N}^k$ of configurations in the natural way: $\mathcal{L}_{\mathcal{A}}(S) \overset{\mathit{def}}{=} \bigcup_{(s,v)\in S} \mathcal{L}_{\mathcal{A}}(s, v)$ and define the language of $\mathcal{A}$ as that of its initial state with all counters zero: $\mathcal{L}(\mathcal{A}) \overset{\mathit{def}}{=} \mathcal{L}_{\mathcal{A}}(s_0, \mathbf{0})$.

We will sometimes denote languages using short-hand "counting expressions". For instance, we write $a^n b^{\leq n}$ for the language $\{a^n b^m \mid n \geq m\}$ over $\Sigma = \{a, b\}$.

Deterministic and finitely-sequential VASSs. A VASS $\mathcal{A} = (\Sigma, Q, \delta, s_0, F)$ is called ε-free if no transition is labelled by ε. It is *deterministic* if it is ε-free and for every pair $(s, a) \in Q \times \Sigma$ there is at most one transition $t = (s, a, e, s') \in \delta$. A VASS is *finitely sequential* if it is the finite union of deterministic VASSs. That is, all transitions from its initial state s_0 are labelled by ε and lead to an initial state of one of finitely many deterministic VASSs.

History-deterministic VASSs. A VASS is *history-deterministic* if one can resolve nondeterministic choices on-the-fly. More formally, consider a function $r : (Q \times \mathbb{N}^k \times \delta)^*(Q \times \mathbb{N}) \times \Sigma \to \delta$ that, given a finite run $\rho_i = (s_0, v_0) \xrightarrow{t_1} (s_1, v_1) \xrightarrow{t_2} \ldots \xrightarrow{t_i} (s_i, v_i)$ and a next letter $a_{i+1} \in \Sigma$, returns a transition $r(\rho_i, a_{i+1}) = t_{i+1} = (s_i, a_{i+1}, e_{i+1}, s_{i+1}) \in \delta$ with $v_i + e_{i+1} \in \mathbb{N}^k$. This yields, for every word $w = a_1 a_2 \ldots \in \Sigma^*$ and initial configuration (s_0, v_0), a unique run in which the ith step $(s_{i-1}, v_{i-1}) \xrightarrow{t_i} (s_i, v_i)$ results from a transition chosen by r. Such a function is called *resolver* if for any input word $w \in \mathcal{L}_{\mathcal{A}}(s_0, v_0)$ the constructed run ρ from initial configuration (s_0, v_0) is accepting. A k-VASS is *history-deterministic* if such a resolver exists.

Language Classes. We denote by k-$\mathcal{D}$, k-$\mathcal{H}$,and k-$\mathcal{N}$ the classes of languages recognised by k-dimensional ε-free deterministic, history-deterministic, and fully non-deterministic VASSs, in the coverability semantics. Similarly, let k-$\mathcal{D}^0$, k-$\mathcal{H}^0$, and k-$\mathcal{N}^0$ denote the classes of languages recognised by k-dimensional ε-free deterministic, history-deterministic, and fully non-deterministic VASSs, in the reachability semantics. Finally, define k-$\mathcal{H}_\varepsilon$,k-$\mathcal{N}_\varepsilon$ k-$\mathcal{H}^0_\varepsilon$, and k-$\mathcal{N}^0_\varepsilon$, as above but without the restriction to ε-free systems. When dropping the parameter k we refer to the union over all dimensions k. For instance, $\mathcal{H} \overset{\mathit{def}}{=} \bigcup_{k\in\mathbb{N}} k$-$\mathcal{H}$.



Language	Definition	Alphabet	Page
L_1	$a^n b^{\leq n} + a^* b^* c$	$\{a, b, c\}$	5
L_2	$a^n b^{\geq n} \#$	$\{a, b, \#\}$	6
L_3	$(a+b)^* a^n b^{\leq n}$	$\{a, b\}$	6
L_4	$a^n b^{\leq n}$	$\{a, b\}$	7
L_5	$a^n b^n$	$\{a, b\}$	7
L_6	$bin(n)\#0^{\leq n}\#$, where $bin(n)$ is n in binary.	$\{0, 1, \#\}$	8
L_7	$a^n b^{\leq n} \#$	$\{a, b, \#\}$	8

Figure 1 Comparison of expressive power of VASS and H-VASS language classes, with and without silent transitions, in reachability and coverability semantics. A solid arrow A → B indicates strict inclusion $\mathsf{A} \subsetneq \mathsf{B}$, with a separating language denoted on the edge. A red/dashed line indicates pair-wise incomparability, with the separating languages denoted. Dotted arrows indicate a special case.

3 Expressiveness

We consider the hierarchy of language classes recognised by vector addition systems, varying definitions in three directions: the degree of non-determinism, reachability vs coverability acceptance, and with/without ε-transitions.

The situation is depicted in Figure 1. We start by looking at the classes defined by ε-free systems (in Section 3.1) before discussing the effect of ε-transitions (in Section 3.2) and following this up with a comparison with finitely-sequential VASS (in Section 3.3).

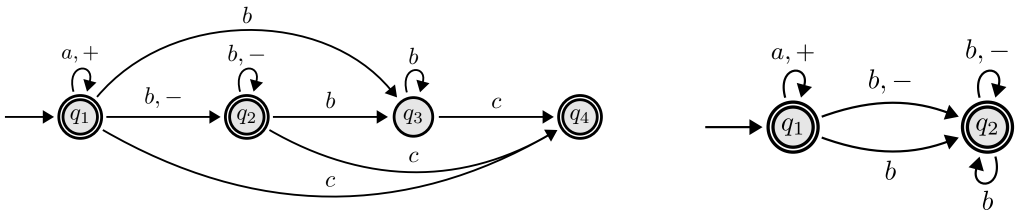
3.1 Separating determinism, history-determinism and non-determinism

In terms of the classes of languages they define, history-deterministic VASSs are strictly more expressive than deterministic ones, and in turn strictly subsumed by fully non-deterministic ones. The following theorem states this formally. Its proof is split into Lemmas 2–5.

▶ **Theorem 1.** *For all $k \geq 1$, we have k-$\mathcal{D} \subsetneq k$-$\mathcal{H} \subsetneq k$-$\mathcal{N}$ and k-$\mathcal{D}^0 \subsetneq k$-$\mathcal{H}^0 \subsetneq k$-$\mathcal{N}^0$.*

▶ **Lemma 2.** *$L_1 \overset{def}{=} a^n b^{\leq n} + a^* b^* c \in 1$-$\mathcal{H} \setminus \mathcal{D}$.*

Proof. L_1 can be recognised by the 1-H-VASS depicted in Figure 2a. Note that the VASS is HD: the only non-deterministic choice is whether to go to q_2 or q_3 on b, for which the resolver must always choose q_2 if available (if the counter is non-zero). The choice of resolver is unique as going to q_3 unnecessarily is not language maximal.



(a) A 1-H-VASS recognising L_1.

(b) A 1-H-VASS^0 recognising L_2.

Figure 2 Transitions labelled with $+$ increment the counter by 1, and those labelled by $-$ decrement the counter by 1 and otherwise have no effect on the counter.

For a contradiction, suppose L_1 is accepted by a k-D-VASS with n states. Since $w_{n+1} = a^{n+1}b^{n+1} \in L_1$ the run is accepted. There exists $i < j \leq n+1$ such that $a^{n+1}b^i$ is in state q with counter vector $v \in \mathbb{N}^k$ and $a^{n+1}b^j$ is also in state q with counter vector $v' \in \mathbb{N}^k$. Since $a^{n+1}b^i \in L_1$, we have that state q is accepting.

Suppose $v' - v \geq \mathbf{0}$, then $a^{n+1}b^{i+(j-i)n} \notin L_1$ is accepted. Therefore there exists a dimension such that $v' - v$ is negative. Hence for some ℓ we have $a^{n+1}b^{i+(j-i)\ell}$ is a dead run. Hence it cannot accept $a^{n+1}b^{i+(j-i)\ell}c \in L_1$. ◀

▶ **Lemma 3.** $L_2 \overset{def}{=} a^n b^{\geq n} \in 1\text{-}\mathcal{H}^0 \setminus \mathcal{D}^0$

Proof. L_2 is recognised by the H-VASS^0 depicted in Figure 2b. On b the resolver can choose between decrementing the counter and no effect, the resolver will always choose to decrement whenever the counter is non-zero.

We have $L_2 \notin \mathcal{D}^0$. Suppose a D-VASS^0 with n states exists, consider the run on the word $w_{n+1} = a^{n+1}b^{n+1} \in L_2$. There exists two prefixes of the run in which $a^{n+1}b^i$ and $a^{n+1}b^j$ revisit a state, and so the system cycles through states on extension of $a^{n+1}b^i$ with b^*. Thus, in order to accept $w_{n+1}b^i$ for all i the automaton must visit only accepting states throughout the cycle. Since $a^{n+1}b^i \notin L_2$ the counter must be non-zero, but zero at $u = a^{n+1}b^{i+(j-i)n}$ since $u \in L$, thus the effect of the cycle is decreasing on some counter, there must exist $k > n$ such that the run is dead on $a^{n+1}b^{i+(j-i)k}$. This is a contradiction as $a^{n+1}b^{i+(j-i)k} \in L_2$. ◀

▶ **Lemma 4.** $L_3 \overset{def}{=} \{a, b\}^* a^{n>0} b^{\leq n} \in 1\text{-}\mathcal{N} \setminus \mathcal{H}$.

Proof. L_3 can be accepted a 1-N-VASS, which non-deterministically guesses the start of the last a^*b^* block and accepts if there are fewer b's than a's.

We show that $L_3 \notin \mathcal{H}$. Suppose for contradiction there is a k-H-VASS with $|Q|$ states, $\|\delta\|$ the largest effect on a counter in any transition and a resolver r.

Consider a sequence of accepted words $w_\ell = w_{\ell-1}a^{m_\ell}b^{m_\ell}$, with w_0 the empty word, where m_ℓ is large enough so that there exist $r_{\ell,1} < r_{\ell,2} \leq m_\ell$, such that the run given by the resolver r on $w_{\ell-1}a^{r_{\ell,1}}$ has configuration (q_ℓ, v_ℓ) and $w_{\ell-1}a^{r_{\ell,2}}$ has (q_ℓ, u_ℓ), with $u_\ell \geq v_\ell$. In other words, whilst reading a^{m_ℓ}, the run encounters a cycle on state q_ℓ which does not strictly decrease any counter value. This occurs due to Dickson's lemma and depends on $|Q|, \|\delta\|, k$ and $m_1, \ldots, m_{\ell-1}$. This gives an inductive way to build words w_ℓ consisting of ℓ blocks of as and bs such that each a-block visits a non-decreasing cycle. We consider the word w_n for $n = 2^k + 1$ and the run ρ on w_n given by the resolver.

Given a vector $v \in \mathbb{N}^k$, we define $support(v) = \{i \mid v_i \neq 0\}$. Since there are n blocks of a in w_n, each of which has a non-decreasing cycle (q_ℓ, u_ℓ) and (q_ℓ, v_ℓ), for $\ell \in \{1, \ldots, n\}$. However, there are $2^k + 1$ possible choices for $support(u_\ell - v_\ell)$. Therefore, there exists $\ell < \ell'$ such that $support(u_\ell - v_\ell) = support(u_{\ell'} - v_{\ell'})$. In other words, there are two a-blocks which have a

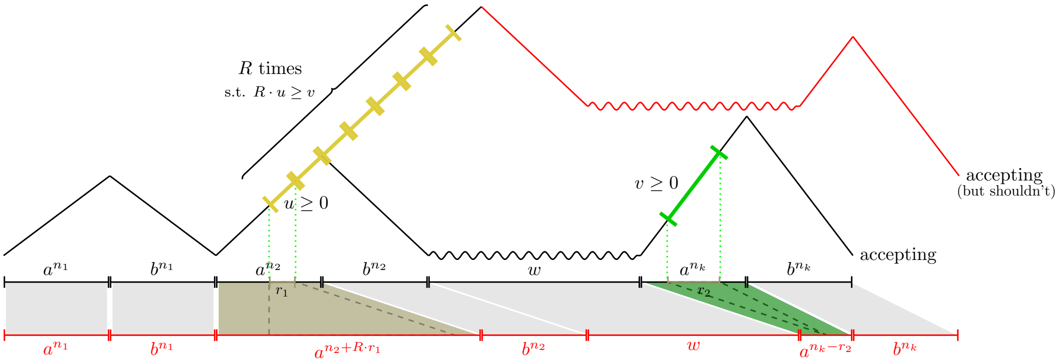


Figure 3 Proof that $L_3 \notin \mathcal{H}$ (Lemma 4). For two cycles of lengths r_1, r_2 chosen in different a^*-blocks with effects $u, v \geq \mathbf{0}$ and *support*$(u) =$ *support*(v), repeating the first cycle and removing the second one constructs an accepting run on a word $\notin L_3$.

non-decreasing cycle such that the effect of the cycles have the same support. Let $R \in \mathbb{N}$ be such that $R(u_\ell - v_\ell) \geq u_{\ell'} - v_{\ell'}$, which exists since $support(u_\ell - v_\ell) = support(u_{\ell'} - v_{\ell'})$ and $u_\ell - v_\ell \geq \mathbf{0}$ and $u_{\ell'} - v_{\ell'} \geq \mathbf{0}$.

Let u be the word such that $w_{\ell'-1} = w_\ell u$, i.e, the part between the ℓth b-block and ℓ'th a-block. Consider the word $w' = w_{\ell-1} a^{m_\ell + R(r_{\ell,2} - r_{\ell,1})} b^{m_\ell} u a^{m_{\ell'} - (r_{\ell',2} - r_{\ell',1})} b^{m_{\ell'}}$. The word w' is therefore obtained by adding $R(r_{\ell,2} - r_{\ell,1})$ many a's in the ℓth a-block and removing $(r_{\ell',2} - r_{\ell',1})$ many a's from the ℓ'th a-block. Note that $w' \notin L_3$, since the last block has more b's than a's. We will show that there is an accepting run on w', by modifying the resolver run on w'_ℓ.

Let $\rho_{\ell'}$ be the run on $w_{\ell'}$ given by the resolver r. We consider the run ρ' where we take the cycle between (q_ℓ, v_ℓ) and (q_ℓ, u_ℓ) an additional R times in the ℓ-th a-block, but removes the cycle between $(q_{\ell'}, v_{\ell'})$ and $(q_{\ell'}, u_{\ell'})$. We show that ρ' is a run on w'. To see this, we must verify that no counter drops below zero in ρ'. Note that the runs $\rho_{\ell'}$ and ρ' are the same till the prefix $w_{\ell-1} a^{r_{\ell,2}}$ after which it reaches the configuration (q_ℓ, u_ℓ). Then it does R additional cycles which results in the configuration $(q_\ell, u_\ell + R(u_\ell - v_\ell))$. From this point ρ' follows the same sequence of transitions as $\rho_{\ell'}$ till it reads the prefix up to $w_{\ell'-1} a^{r_{\ell',1}}$ ending up in the configuration $(q_{\ell'}, v_{\ell'} + R(u_\ell - v_\ell))$. Since $v_{\ell'} + R(u_\ell - v_\ell) \geq v_{\ell'} + (u_{\ell'} - v_{\ell'}) = u_{\ell'}$, ρ' can follow the suffix of the run $\rho_{\ell'}$ from $(q_{\ell'}, u_{\ell'})$ on $a^{m_{\ell'} - r_{\ell',2}} b^{m_{\ell'}}$, which ends in the same state as $\rho_{\ell'}$ with a non-zero counter value. This is a contradiction as we get a accepting run on $w' \notin L_3$. We conclude that there is no k−H-VASS that recognises the language L_3. ◀

▶ **Lemma 5.** $L_4 \overset{def}{=} a^n b^{\leq n} \in 1\text{-}\mathcal{N}^0 \setminus \mathcal{H}^0$

Proof. In the non-deterministic case reachability semantics can recognise $L_4 \in 1\text{-}\mathcal{N}^0$: On a, non-deterministically choose either to increment by 1 or not, guessing ahead of time how many b's will be seen. On b, the machine moves to a new state and counts down, preventing more b's than the guessed number.

However L_4 cannot be recognised with history-determinism. To see this, observe that since $a^n \in L_4$ all the counters must be zero after reading a^n, then, for n larger than the number of states, the machine cannot distinguish $a^n b^n \in L_4$ and $a^n b^{n+1} \notin L_4$. ◀

3.2 Silent transitions

First observe that $L_5 \overset{def}{=} a^n b^n$ can be recognised with reachability semantics (even $\mathcal{D}^0$), but cannot be recognised under coverability semantics (even $\mathcal{N}_\varepsilon$). On the other hand $L_4 = a^n b^{\leq n}$ can be recognised by coverability semantics (even $\mathcal{D}$), but cannot be recognised by $\mathcal{H}^0_\varepsilon$, thus

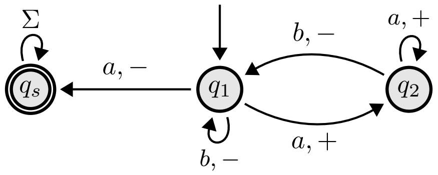


Figure 4 A 1-H-VASS automaton with language $L_8 = \mathcal{L}(q_1, 0)$ that is not finitely sequential. The automaton reads blocks of a's followed by blocks of b's. If some block of a's is followed by fewer b's then the automaton can read anything after the next a. If every block is followed by the same number of a's and b's then it must read another block of the form $a^n b^n$ or $a^n b^{<n}$. The language is thus $L_8 = \bigcup_{k=0}^{\infty} a^{n_0} b^{n_0} \ldots a^{n_{k-1}} b^{n_{k-1}} a^{n_k} b^{<n_k} a\Sigma^*$.

together L_4 and L_5 show pairwise incomparability between reachability and coverability semantics for deterministic and history-deterministic systems. However, if the languages have an end marker then coverability acceptance can be turned into reachability acceptance (with ε-transitions) as ε-transitions can be used to take the counters to zero at the end marker.

The separation between $\mathcal{N}$ and $\mathcal{N}_\varepsilon$ is due to [13] for which $L_6 \overset{def}{=} bin(n)\#0^{\leq n}\# \in \mathcal{N}_\varepsilon \setminus \mathcal{N}$, where $bin(n)$ is the binary representation of $n \in \mathbb{N}, n > 0$ in $1\{0,1\}^*$. This language cannot be recognised without ε transitions (see full version [7] or [13] for details). We observe that the same language separates $\mathcal{H}$ and $\mathcal{H}_\varepsilon$, as the 2-VASS of [13] recognising L_6 is in fact history-deterministic. However, in dimension 1, the two classes collapse:

▶ **Lemma 6.** $1\text{-}\mathcal{H} = 1\text{-}\mathcal{H}_\varepsilon$.

While in coverability semantics, the presence of ε-transitions separates languages recognised by k-H-VASS and k-H-VASS$_\varepsilon$ only for dimensions $k \geq 2$, in reachability semantics the separation occurs already in dimension 1: $L_7 \overset{def}{=} a^n b^{\leq n}\#$ is in $\mathcal{H}^0_\varepsilon$ but not in $\mathcal{H}^0$.

3.3 Comparison with Finitely Sequential VASS

Recall that finitely sequential VASS are the union of finitely many D-VASS. In Lemma 8 we show that language of a finite union of history-deterministic VASS is also history-deterministic. In particular, the deterministic VASSs comprising the finitely sequential VASS are themselves history-deterministic, so any finitely sequential VASS has an equivalent history-deterministic VASS recognising the same language. On the other hand, we show that history-deterministic VASS with coverability acceptance are strictly more powerful:

▶ **Lemma 7.** *There exists a language in* $1\text{-}\mathcal{H}$ *that is not finitely sequential.*

Proof. Consider the language $L_8 \overset{def}{=} \mathcal{L}(q_1, 0)$ of the VASS depicted in Figure 4. Observe that it is history-deterministic: when reading a at state q_1, the resolver goes to q_s if possible. This choice is language-maximal and there is no other non-determinism to resolve.

We show the language is not finitely sequential. Suppose for contradiction the language is accepted by a finitely sequential VASS that is the union of k many D-VASS, each with at most m states. We consider the word $a^{m+1}b^{m+1}$, which can be extended into an accepting word in L_8, and thus is alive in some D-VASS. For D-VASS in which the run is still alive, reading this word goes through a cycle in the run while reading a^{m+1} and similarly also whilst reading b^{m+1}.

Let $c_1, \ldots, c_k$ be the lengths of these cycles while reading a's in each D-VASS respectively, $d_1, \ldots, d_k$ be the lengths of the cycles reading b's, and fix $C = \prod_{i \leq k} c_i$ and $D = \prod_{i \leq k} d$. Observe that, for every x, for each D-VASS the same state is reached after reading a^{m+1+xC}. Similarly, for any y, the same state is reached after reading $a^{m+1+xC}b^{m+1+yD}$. In particular, fix words $w = a^{m+1+CD}b^{m+1+CD}$ and the words $u = a^{m+1+CD}b^{m+1+(C-1)D}$.

Observe that after reading ua, the system in Figure 4 can be in state q_s and therefore, any extension of ua is accepted. However, the automaton of Figure 4 can only reach state q_2 on wa and so, for any $z \in \mathbb{N}, i \geq 1$, $wa^z b^{z+i} \notin L_8$. Consider this word for $z = m+1$. Since there is a cycle somewhere while reading b^z, then when reading more b's the automaton visits only states on that cycle. Since $wa^z b^{z+i} \notin L_8$ for $i \geq 1$ either every state on the cycle is non-accepting, or the cycle has a negative effect on at least one counter and therefore becomes unavailable for large enough i.

Recall, in M both wa and ua are in the same control location in each constituent D-VASS, and thus for any $v \in \Sigma^*$ we have wav and uav reach the same control locations (or possibly the run is dead). However, for every z, i, there is some D-VASS in which the word $ua^z b^{z+i}$ is accepting. However, we have argued that for every D-VASS, for sufficiently large i, the run on $wa^z b^{z+i}$ is stuck in a rejecting cycle, or a cycle in which the counter is decreasing. Thus for sufficiently large i, in every D-VASS, either the run on $ua^z b^{z+i}$ is also dead or in a rejecting cycle, which contradicts $ua^z b^{z+i} \in L_8$. ◀

4 Closure Properties

We take a look at closure properties of the classes $\mathcal{H}$ and $\mathcal{H}^0$ recognised by history-deterministic VASSs in coverability and reachability semantics, respectively.

Union closure (of $\mathcal{H}$ and $\mathcal{H}^0$) and closure under intersection (for $\mathcal{H}$) can be shown using a straightforward product construction at the cost of increasing the dimension.

▶ **Lemma 8.** *Let $L \in k\text{-}\mathcal{H}$ and $L' \in k'\text{-}\mathcal{H}$. Then $L \cup L' \in (k+k')\text{-}\mathcal{H}$ and $L \cap L' \in (k+k')\text{-}\mathcal{H}$. Let $L \in k\text{-}\mathcal{H}^0$ and $L' \in k'\text{-}\mathcal{H}^0$. Then $L \cap L' \in (k+k')\text{-}\mathcal{H}^0$.*

A naïve product of the two systems recognising L and L' does *not* work for showing the union closure of $\mathcal{H}^0$ because here, acceptance requires all counters to be zero even for inputs that are only in one of the two languages (note the absence of ε-transitions). Indeed, $\mathcal{H}^0$ are not closed under union, as witnessed by $L_9 \overset{def}{=} a^n b^n \cup a^n b^{2n}$ not being in $\mathcal{H}^0$ (see full version [7]).

Taking a direct product yields a H-VASS that may not be optimal in terms of the number of counters and in general, increasing the dimension is not avoidable. For instance, the languages $L_{10} \overset{def}{=} a^n b^{\leq n} c^* \cup a^n b^* c^{\leq n}$ and $L_{11} \overset{def}{=} a^n b^{\leq n} c^* \cap a^n b^* c^{\leq n}$ are not in 1-$\mathcal{H}$, while the individual component languages are. Similarly, the language $L_{12} \overset{def}{=} a^n b^n c^* \cap a^n b^* c^n = a^n b^n c^n$ witnesses non-closure of 1-$\mathcal{H}^0$ under intersection.

The theorems below summarise our findings regarding closure properties of history-deterministic classes. Full proofs are in the full version [7].

▶ **Theorem 9.** *$\mathcal{H}$ is closed under union, intersection and inverse homomorphisms. It is not closed under complementation, concatenation, homomorphisms, iteration, nor commutative closure.*

▶ **Theorem 10.** *$\mathcal{H}^0$ is closed under intersection and inverse homomorphisms. It is not closed under union, complementation, concatenation, homomorphisms, iteration, nor commutative closure.*

5 Decision Problems

In this section we consider decision problems related to history-determinism: checking if a given N-VASS is history-deterministic, HD definability (as well as regularity) of its recognised language, and language inclusion between HD VASSs.

Prakash and Thejaswini [32] showed that in dimension 1 (and for coverability semantics), checking HDness and inclusion is decidable in PSPACE by reduction to simulation preorder [18]. This can be generalised slightly as follows.

▶ **Theorem 11.** *Language inclusion $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$ is decidable for any* 1*-H-VASS $\mathcal{A}$ and for any N-VASS $\mathcal{B}$.*

Proof. By Theorem 19 in [32], for any 1-H-VASS, one can effectively construct a language equivalent deterministic one-counter automaton (DOCA; a 1-VASS with zero-testing transitions). DOCA can be complemented [36] and so the inclusion question is equivalent to the emptiness (reachability) of $\overline{\mathcal{A}} \times \mathcal{B}$, a VASS with one zero-testable counter, which is decidable [34]. Note this result is independent of the number of counters of B. ◀

We continue to show that in higher dimensions, these questions are undecidable. Throughout this section, when we show undecidability for ε-free VASS, the result naturally also applies for superclass with silent transitions. Our constructions proving this are similar, yet require subtle differences, and are all based on weakly simulating two-counter machines [30]. Let us recall these in a suitable syntax first.

▶ **Definition 12.** *A two-counter Minsky machine (2CM) $M = (Q, q_0, q_h, \delta)$ consists of a finite set of states Q, including a distinguished starting and final state q_0, q_h, respectively, as well as a finite set of transitions $\delta \subseteq Q \times \Gamma \times Q$, where $\Gamma = \{inc_1, inc_2, dec_1, dec_2, ztest_1, ztest_2\}$ are the operations on the counters*[1]*.*

A configuration of M is an element of $Q \times \mathbb{N}^2$, comprising the current state and the value of the two counters. For every state q either:

1. *There is only one transition of the form (q, inc_i, q'). This allows to move from state q to q', increment counter i by one and leaves the other counter untouched; or*
2. *There are exactly two transitions from q, of the form $(q, ztest_i, q')$ and (q, dec_i, q''). The former allows to move to q' without changing the counters, but only if counter i has value 0. The latter allows to move from q to q'' and decrease counter i, and leaves the other counter unchanged.*

Notice that from any configuration there is exactly one possible successor configuration. We can therefore speak of *the* run of M, and its sequence of counter operations, from the initial configuration $(q_0, 0, 0)$. We say that M *terminates* if its run visits the final state q_h. W.l.o.g., we can assume that both counters have value 0 whenever M terminates.

Deciding whether a given 2CM terminates is undecidable [30]. An easy consequence, and the basis for our construction for regularity, is the undecidability of checking finiteness of the reachability set for a given 2CM.

▶ **Lemma 13.** *It is undecidable to check, for given 2CM M, if its run visits infinitely many different configurations.*

[1] Readers may be more familiar with an instruction of the form if $C_i = 0$ goto q_ℓ else goto q_k, this can be simulated by a ztest$_i$ to q_ℓ and a decrement followed by an increment to q_k.

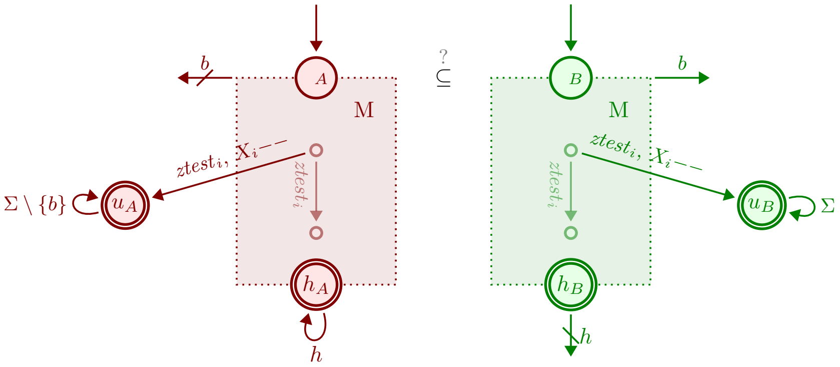


Figure 5 The 2-VASSs A (in red) and B (in green) both include a copy of, and weakly simulate, a given 2CM M. For any zero-testing operation ztest_i in M both can go to a sink state if counter i is in fact non-zero, reading the letter ztest_i and decreasing the VASS counter i, as indicated by the effect vector $X_i{-}{-}$. The extra letter b ensures that $\mathcal{L}(B) \not\subseteq \mathcal{L}(A)$; Only A can accept words that consist of valid sequences of 2CM operations and that end in the letter h.

5.1 Checking HDness and Inclusion

We focus on the questions of whether a given VASS is history-deterministic, and whether language inclusion holds for two languages given by H-VASS. For languages of finite words these two decision problems are intrinsically linked due to the connection with language maximal resolvers.

▶ **Lemma 14.** *For a given 2CM M one can construct two history-deterministic 2-VASSs with initial states s_A and s_B, respectively, so that $\mathcal{L}(s_A, 0) \subseteq \mathcal{L}(s_B, 0)$ if, and only if, the unique valid run of M never reaches a halting state.*

Proof. Suppose we are given 2CM M with designated initial and halting states s and h, respectively, and let Γ denote the set of counter operations. W.l.o.g., there is exactly one valid sequence of counter operations that is either infinite or finite. We define two 2-VASSs A and B over the alphabet $\Sigma = \Gamma \uplus \{b, h\}$. These are just copies of, and just weakly simulate the machine M: For every state q of M, there are states q_A and q_B; For every transition $q \xrightarrow{\gamma} q'$ of M, there are corresponding edges $q_A \xrightarrow{\gamma} q'_A$ and $q_B \xrightarrow{\gamma} q'_B$ that read the letter γ and manipulates the counter accordingly: if $\gamma = \text{inc}_i$ (or dec_i) then counter i is incremented (or decremented, respectively). If $\gamma = \text{ztest}_i$ then counter i remains as is. The only accepting states so far are h_A and h_B, corresponding to the designated halting state of M.

Additionally, for every zero-testing transition $q \xrightarrow{\text{ztest}_i} q'$ in M, both A and B have a transition from state q that decreases counter i and goes to a new, accepting, sink state u with language $\supseteq (\Gamma \cup \{h\})^*$. This way, both systems will accept any word that prescribes a run of M that contains a "counter cheat", meaning that the word contains operation ztest_i but the run of M so far ends in a configuration where counter i is not zero.

We now modify the systems A and B so that they differ in two ways:

1. the halting state h_A of A admits a h-labelled step (to itself) but s_B does not.
2. All states in B have b-labelled steps (to the accepting sink u_B) but none of As states do.

See Figure 5 for a depiction of the constructed 2-VASS.

Notice that $\mathcal{L}(B) \not\subseteq \mathcal{L}(A)$ by design, because no word containing the letter b can be accepted by A. Notice that both A and B are indeed history-deterministic: the only choices to be resolved are upon reading a zero-testing letter ztest_i from a configuration where the corresponding counter i is not zero. In any such case, moving to the sink is language maximal.

It remains to argue that $\mathcal{L}(s_A) \not\subseteq \mathcal{L}(s_B)$ if, and only if, M has a finite run from an initial configuration to its final state. Indeed, if M terminates via a sequence $\rho = e_0 e_1, \ldots e_k$, then $\mathcal{L}(A)$ contains the word $\rho \cdot h$. Since this run does not contain "cheats" nor letters b, the system B cannot possibly reach the winning sink u_B and therefore not accept. Conversely, if M does not terminate, then any word $\rho \in \Gamma^* \cdot h$ accepted by A must prescribe a run of M that contains a cheat. Say $\rho = \rho_1 \cdot \mathrm{ztest}_i \cdot \rho_1 \cdot h$. But then, B will be able to reach the sink u_B after reading the prefix $\rho_1 \cdot \mathrm{ztest}_i$ and thus accept. ◀

The construction in the previous lemma works both in coverability and reachability semantics (note that we assume that a 2CM terminates with counters at 0). The next two theorems are direct consequences and again hold for coverability and reachability semantics.

▶ **Theorem 15.** *Checking language inclusion is undecidable for* 2*-HD VASSs.*

▶ **Theorem 16.** *It is undecidable to check if a given* 2*-VASS is history-deterministic.*

Proof. By reduction from 2CM termination: Construct the two systems A and B as given by Lemma 14 and add one new initial state s that, upon reading some letter b can move to the initial state s_A of A or s_B of B. The so-constructed system is HD iff $\mathcal{L}(s_A) \subseteq \mathcal{L}(s_B)$, which is true iff M does not terminate. ◀

5.2 Checking HDness of VASS Languages

We turn to showing undecidability of *language* history-determinism, i.e., the question if for a given VASS there exists an equivalent history-deterministic VASS. We start with the more interesting and involved case, for the coverability semantics (Theorem 17) and present an easier construction for reachability (Theorem 18) afterwards.

We give a proof by reduction from the 2CM halting problem, combining the constructions to show the non-HDness of $L_3 = (a,b)^* a^n b^{\leq n}$, (Lemma 4) and the proof of [23] that checking regularity for N-VASS languages is undecidable.

▶ **Theorem 17.** *It is undecidable to check if $\mathcal{L}(\mathcal{A}) \in \mathcal{H}$ holds for a given N-VASS $\mathcal{A}$.*

Proof. By reduction from the 2CM halting problem. For a given 2CM M with states Q_M and counter operations $\Gamma = \{\mathrm{inc}_1, \mathrm{inc}_2, \mathrm{dec}_1, \mathrm{dec}_2, \mathrm{ztest}_1, \mathrm{ztest}_2\}$ we construct a 3-VASS $\mathcal{A} = (\Sigma, Q, \delta, s_0, F)$ so that $\mathcal{L}(\mathcal{A})$ is history-deterministic iff the faithful run of M is finite.

We refer to the three counters as X_1, X_2, X_3 and write $X_i{-}{-}$ and $X_i{+}{+}$ for the effects of (VASS) transitions that decrement/increment counter i only.

The construction. $\mathcal{A}$ uses the alphabet $\Sigma = \Gamma \cup \{a, b\}$, consisting of counter operations of M and two fresh symbols. The control states of $\mathcal{A}$ mimic those of M, except that in between any simulated step of M, $\mathcal{A}$ can read a word in a^+b^+: For every state $q \in Q_M$ we introduce states q_{in}, q_{out} and q_{step}. In addition, we add three other states sink, r_1, r_2. We make sink universal by adding self-loops $(s, a, \mathbf{0}, s)$ for every letter $a \in \Sigma$. First we consider the simulation of M.

For every step $q \xrightarrow{\gamma} p$ of M, $\mathcal{A}$ has a transition $t = (q_{out}, \gamma, e, p_{in})$ from q_{out} to p_{in} that reads the letter $\mathit{label}(t) = \gamma$ and manipulates the counter accordingly: if $\gamma = \mathrm{inc}_i$ then $e = X_i{+}{+}$; if $\gamma = \mathrm{dec}_i$ then $e = X_i{-}{-}$; if $\gamma = \mathrm{ztest}_i$ then $e = \mathbf{0}$. In addition, for zero-testing steps $q \xrightarrow{\mathrm{ztest}}_i p$, $\mathcal{A}$ in M, $\mathcal{A}$ contains a decreasing transition $t = (q_{out}, \mathrm{ztest}_i, X_i{-}{-}, \mathit{sink})$ to the universal sink state. From a state q_{in}. There are two possible continuations:

1. Reading a word in a^+b^+ and moving to q_{out}, via transitions $q_{in} \xrightarrow{a,\mathbf{0}} q_{step}$, $q_{step} \xrightarrow{a,\mathbf{0}} q_{step}$, $q_{step} \xrightarrow{b,\mathbf{0}} q_{out}$ and $q_{out} \xrightarrow{b,\mathbf{0}} q_{out}$.
2. Reading a word in $a^n b^{\leq n}$ and stopping. For this, there are transitions $q_{in} \xrightarrow{a,X_3{+}{+}} r_1$, $r_1 \xrightarrow{a,X_3{+}{+}} r_1$, $r_1 \xrightarrow{b,X_3{-}{-}} r_2$ and $r_2 \xrightarrow{b,X_3{-}{-}} r_2$.

The accepting states of $\mathcal{A}$ are $F = \{r_2, \mathit{sink}\}$. Its initial state is $s_0 = q_{out}$, where $q \in Q_M$ is the initial state of M.

The recognised language. The language of the constructed 3-VASS $\mathcal{A}$ contains sequences of instructions of M interspersed with blocks of the form a^+b^+. Let's call a sequence $\gamma_1\gamma_2\dots\gamma_k \in \Gamma^*$ of operations in M *faithful* if for all $i \leq k$, γ_i is the ith instruction in the run of M from its initial configuration $(q, 0, 0)$. Clearly, for any k less or equal to the length of the run of M, there is a unique faithful sequence ρ_k of length k. Define $\mathsf{Correct}_k \overset{def}{=} \gamma_1(a^+b^+)\gamma_2(a^+b^+)\gamma_3\dots(a^+b^+)\gamma_k$ where $\gamma_1\gamma_2\dots\gamma_k = \rho_k$. Let $\mathsf{Incorrect}_k \subseteq \Sigma^*$ contain exactly all words $w\gamma \in \Sigma^* \setminus \mathsf{Correct}_k$ where $w \in \mathsf{Correct}_{k-1}$ and $\gamma \in \{\mathrm{ztest}_1, \mathrm{ztest}_2\}$. That is, words whose projection into the operations of M is faithful up to step $k-1$ but that contain an incorrect zero-test at step k.

Observe that if the faithful sequence of length k takes M to (q, C_1, C_2) then $\mathcal{A}$ can read any word in $\mathsf{Correct}_k$ and every run on such a word leads to the configuration $(q_{in}, C_1, C_2, 0)$. Such a run of $\mathcal{A}$ can be extended in two ways to reach an accepting state. Either by reading a word in $a^n b^{\leq n}$ to reach r_2, or by continuing on the run of M and eventually erroneously reading a ztest_i to reach sink. We can therefore write the language of $\mathcal{A}$ as

$$\mathcal{L}(\mathcal{A}) \quad = \quad \bigcup_{k \geq 0} \mathsf{Correct}_k \cdot (a^n b^{\leq n}) \quad \cup \quad \bigcup_{k \geq 0} \mathsf{Incorrect}_k \cdot \Sigma^*$$

HDness. We show that if M terminates, meaning its run has some length $k \in \mathbb{N}$, then $\mathcal{L}(\mathcal{A})$ is history-deterministic. Observe that for every $0 \leq i \leq k$, both languages $\mathsf{Correct}_i$ and $\mathsf{Incorrect}_i$ are regular. We can concatenate a DFA recognising the former with a 1-H-VASS for $a^n b^{\leq n}$ to construct an 1-H-VASS recognising $\mathsf{Correct}_i \cdot (a^n b^{\leq n})$. Observe that $\mathsf{Incorrect}_i$, $i > k+1$ is empty. Now, $\mathcal{L}(\mathcal{A})$ is the finite union of k many 1-H-VASS languages (and a regular language) and therefore recognisable by a k-dimensional H-VASS.

It remains to show that if the run of M is infinite, then $\mathcal{L}(\mathcal{A})$ is not in k-$\mathcal{H}$, for any k. Our proof mirrors the proof of Lemma 4, except that we interleave $\{a, b\}$-blocks with the faithful operations of M. Suppose towards a contradiction that there exists a k-H-VASS $\mathcal{B}$ with states $Q_{\mathcal{B}}$ and let $\rho = \gamma_1\gamma_2, \dots \in \Gamma^\omega$ denote the infinite run of M. That is, every length-i prefix ρ_i is faithful. Consider a sequence $(w_n)_{n \geq 0}$ of words in $\mathcal{L}(B)$ such that $w_0 = \varepsilon$ and otherwise $w_\ell = w_{\ell-1}\gamma_\ell a^{m_\ell} b^{m_\ell}$ with m_ℓ large enough so that the resolved run on w_ℓ contains a non-decreasing cycle while reading the last a-block. Say,

$$(s_0, \mathbf{0}) \xrightarrow{w_{\ell-1}\gamma_\ell a^{r_{\ell,1}}} (q_\ell, u_\ell) \xrightarrow{a^{r_{\ell,2}}} (q_\ell, v_\ell)$$

with $u_\ell \leq v_\ell$. This is well-defined by Dickson's Lemma.

Setting $n = |Q_{\mathcal{B}}|2^k + 1$ is sufficiently high so that there must be $\ell < \ell'$ with $q_\ell = q_{\ell'}$ and $\mathit{support}(u_\ell - v_\ell) = \mathit{support}(u_{\ell'} - v_{\ell'})$. Take R be such that $R(u_\ell - v_\ell) \geq (u_{\ell'} - v_{\ell'})$ and let u be the word such that $w_{\ell'-1} = w_\ell u$. Now consider the word

$$w' = w_{\ell-1}\gamma_\ell a^{m_\ell + R(r_{2,\ell})} b^{m_\ell} u \gamma_{\ell'} a^{m_{\ell'} - r_{2,\ell'}} b^{m_{\ell'}}$$

that results from w_n by removing one iteration of the loop in block ℓ' and making up for it by inserting R iterations of the loop in block ℓ. Notice that w' is accepted by the run that follows the resolved run on w_n and repeats the designated loops on the extra letters. However, $w' \notin \mathcal{L}(\mathcal{A})$ because its last $\{a, b\}$-block contains more b's than a's. ◀

Notice that if the given 2CM terminates then our construction produces a history-deterministic VASS where the number of counters corresponds to the length of the terminating run. Therefore it remains open whether the language k-HDness problem is decidable, which ask whether there is an equivalent k-HDVASS for the given language.

The analogous statement for reachability is simpler to prove; by adapting the construction for the regularity problem of Parikh-automata [11], we reduce from the universality problem, which is undecidable in reachability semantics [37, Theorem 10].

▶ **Theorem 18.** *It is undecidable to check if $\mathcal{L}(\mathcal{A}) \in \mathcal{H}^0$ holds for a given N-VASS $\mathcal{A}$.*

Proof. We reduce from the undecidable universality problem for VASS languages in reachability semantics [37, Theorem 10]. The construction is the same as for the regularity problem of Parikh-automata, recently presented in [11]. For an alphabet Σ let $\Sigma_\$ = \Sigma \uplus \{\$\}$ for some fresh symbol $\$ \notin \Sigma$. For two words u, v let $u \otimes v$ be the word $w = (a_1, b_1)(a_2, b_2) \dots (a_k, b_k)$ so that either $u = a_1 a_2 \dots a_k$ and $b_1 b_2 \dots b_k \in v\$^*$ or $v = b_1 b_2 \dots b_k$ and $a_1 a_2 \dots a_k \in u\$^*$.

For two languages $L, L' \subseteq \Sigma^*$ define their *cross-union* $L \circledvee L \subseteq (\Sigma_\$^2)^*$ to be the langugae of words $u \otimes v$ such that $u \in L$ or $v \in L'$. That is, for any word $w \in L \circledvee L$, either the projection into the first components is $\pi_1(w) \in L$ or that into the second components $\pi_2(w) \in L'$.

Recall the language $L_4 = a^n b^{\leq n} \in \mathcal{N}^0 \setminus \mathcal{H}^0$, which is not HD recognisable. To show our claim, let L be some given N-VASS language and consider the language $L_{14} \overset{def}{=} \$ \cdot (L \circledvee \emptyset) \cup \$ \cdot (\emptyset \circledvee L_4)$. This is clearly in $\mathcal{N}^0$. Now, if $L = \Sigma^*$ is universal then $L \circledvee \emptyset$ is universal over $\Sigma_\$^2$ and so $L_{14} = \$(\Sigma_\$^2)^* \in \mathcal{H}^0$ (even, regular). If conversely, suppose L is not universal as witnessed by $w \notin L$, then L_{14} cannot be recognised by any H-VASS^0 for the same reason as $a^n b^{\geq n} \notin \mathcal{H}^0$: suppose it is accepted by some k-H-VASS^0 run on n states and consider run of the resolver on the word $u = \$(w \otimes a^{|w|+n+1}) \in L_{14}$, thus must end with counter $\mathbf{0}$. The extension of u by $(\$, b)^{n+1}$ is also accepting, it must remain at $\mathbf{0}$ and cycle on accepting states. Hence $u(\$, b)^{|w|+n+1} \in L_{14}$ cannot be distinguished from $u(\$, b)^{|w|+n+2} \notin L_{14}$. ◀

5.3 Regularity

We turn to the decision problem of whether a given VASS recognises a regular language. This regularity question is undecidable for general N-VASS [23]. It again turns out that for history-deterministic VASSs, the decidability status of regularity depends on the dimension. For 1-H-VASS, one can effectively construct a language equivalent DOCA [32], for which checking regularity remains decidable [2, 36].

▶ **Theorem 19.** *Given a* 1*-H-VASS $\mathcal{A}$, checking if $\mathcal{L}(\mathcal{A})$ is regular is decidable in* EXPSPACE*.*

Although checking regularity of DOCA is NL-complete, the added complexity here is due to the doubly exponentially large DOCA produced in the reduction. Since 1-H-VASS$_\varepsilon$ can be transformed into 1-H-VASS by Lemma 6, the theorem also holds for 1-H-VASS$_\varepsilon$. We now show undecidability already for dimension 2.

▶ **Theorem 20.** *Given a* 2*-H-VASS $\mathcal{A}$, it is undecidable if $\mathcal{L}(\mathcal{A})$ is regular.*

Proof. By reduction from the finiteness problem for 2CM (Lemma 13). For a given 2CM M we construct a 2-H-VASS whose language will be regular iff M's run visits only finitely many configurations. We make the argument for coverability semantics first.

Let $\rho = \gamma_1 \gamma_2 \dots$ be the faithful run of M and $|\rho| \in \mathbb{N} \cup \{\infty\}$ for its length. Write $\mathit{correct}_k$ for its length-k prefixes and let x_k be 1 plus the sum of both counter-values in the configuration M reaches after reading $\mathit{correct}_k$. Further, wherever $\mathit{correct}_k = \mathit{correct}_{k-1}\mathrm{dec}_i$, define $\mathit{incorrect}_k$ as $\mathit{correct}_{k-1}\mathrm{ztest}_i$.

Consider the language $L = G \uplus B$ over the alphabet $\Sigma = \Gamma \uplus \{a\}$,

$$G \overset{def}{=} \bigcup_{k\geq 0} \left(\rho_k \cdot a^{\leq x_k}\right) \qquad \text{and} \qquad B \overset{def}{=} \bigcup_{k\geq 0} \left(\mathit{incorrect}_k \cdot \Sigma^*\right)$$

G consists of words that describe some length-k prefix of M's run followed by x_k or fewer symbols a; B contains all words describing the run of M up to length-k, followed by an incorrect zero-test, and then anything.

We claim that this language L is recognised by a 2-H-VASS. To see this, again build a VASS that weakly simulates M as done before, for example in the proof of Theorem 17. This will simulate increment and decrement operations faithfully, reading letters inc_i or dec_i, respectively. For any step $q \xrightarrow{\mathrm{ztest}_i} q'$ in M, the VASS $\mathcal{A}$ will have a transition $(q, \mathrm{ztest}_i, \mathbf{0}, q')$ as well as one that reads ztest_i, decreases counter i and leads to a universal state. This allows to accept exactly all words in B. In addition, from any state q of M, $\mathcal{A}$ can move to a new countdown phase: there is a transition $q \xrightarrow{a,\mathbf{0}} c$ to a new, final, control state that can continue to read $a's$ while at least one of the counters remains non-zero. This allows to accept exactly all words in G. Note that the only non-determinism is for letters ztest_i when M's ith counter after reading ρ_i is not zero. In this case, the only language-maximal choice is to move to the universal state. The constructed system is therefore history-deterministic.

To conclude the proof, we argue that L is regular iff ρ visits only finitely many configurations. Indeed, if so, then G is finite because all x_i, $i \leq k$ are bounded, and B is regular because at most k many words $\mathit{incorrect}_k$ exist. So L is the finite union of regular languages and thus regular.

Conversely, suppose that M's run ρ visits infinitely many different configurations. Then in particular, there are infinitely many faithful prefixes ρ_k. Let us assume towards contradiction that L is regular and recognised by a DFA with d many states. We pick a prefix ρ_k so that $x_k > d$ and consider the word $\rho_k a^{x_k} \in L$. While reading the suffix a^m, our DFA must repeat some cycle of length $c \leq d$. But then it must also accept $\rho_k a^{x_k+c} \notin L$ by going through that cycle twice.

The same proof goes through for the reachability semantics if we set $G \overset{def}{=} \bigcup_{k\geq 0} \left(\rho_k \cdot a^{=x_k}\right)$ and $B \overset{def}{=} \bigcup_{k\geq 0} \left(\mathit{incorrect}_k \cdot \Sigma^{\geq x_k - 1}\right)$. Then again, if the run of M visits finitely many configurations then both G and B are regular. Otherwise G is not regular. The extra symbols at the end (of words in G and B) allow a run of the VASS $\mathcal{A}$ to decrease the counters to 0 and accept (and therefore to conclude that language $L = G \uplus B$ is in 2-$\mathcal{H}^0$). ◀

References

1 Stanislav Böhm, Stefan Göller, Simon Halfon, and Piotr Hofman. On büchi one-counter automata. In *International Symposium on Theoretical Aspects of Computer Science*, volume 66 of *LIPIcs*, pages 14:1–14:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.STACS.2017.14.

2 Stanislav Böhm, Stefan Göller, and Petr Jancar. Bisimulation equivalence and regularity for real-time one-counter automata. *Journal of Computer and System Sciences*, 80(4):720–743, 2014. doi:10.1016/j.jcss.2013.11.003.

3 Udi Boker and Karoliina Lehtinen. Good for games automata: From nondeterminism to alternation. In *International Conference on Concurrency Theory*, volume 140 of *LIPIcs*, pages 19:1–19:16, 2019.

4 Udi Boker and Karoliina Lehtinen. History Determinism vs. Good for Gameness in Quantitative Automata. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 213 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FSTTCS.2021.38.

5 Udi Boker and Karoliina Lehtinen. Token games and history-deterministic quantitative automata. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 120–139. Springer International Publishing, 2022.

6 Sougata Bose, Thomas A. Henzinger, Karoliina Lehtinen, Sven Schewe, and Patrick Totzke. History-deterministic timed automata are not determinizable. In *International Workshop on Reachability Problems*, 2022.

7 Sougata Bose, David Purser, and Patrick Totzke. History-deterministic vector addition systems. *CoRR*, abs/2305.01981, 2023. doi:10.48550/arXiv.2305.01981.

8 Thomas Colcombet. The theory of stabilisation monoids and regular cost functions. In *International Colloquium on Automata, Languages and Programming*, pages 139–150, 2009.

9 Wojciech Czerwinski and Piotr Hofman. Language inclusion for boundedly-ambiguous vector addition systems is decidable. In *International Conference on Concurrency Theory*, volume 243 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.16.

10 Wojciech Czerwinski and Lukasz Orlikowski. Reachability in vector addition systems is ackermann-complete. In *Annual Symposium on Foundations of Computer Science*, pages 1229–1240. IEEE, 2021. doi:10.1109/FOCS52979.2021.00120.

11 Enzo Erlich, Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen, and Martin Zimmermann. History-deterministic parikh automata. In *International Conference on Concurrency Theory*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

12 Olivier Finkel and Michal Skrzypczak. On the topological complexity of ω-languages of non-deterministic Petri nets. *Information Processing Letters*, 114(5):229–233, 2014. doi:10.1016/j.ipl.2013.12.007.

13 Sheila A. Greibach. Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science*, 7:311–324, 1978. doi:10.1016/0304-3975(78)90020-8.

14 Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen, and Martin Zimmermann. A Bit of Nondeterminism Makes Pushdown Automata Expressive and Succinct. In *International Symposium on Mathematical Foundations of Computer Science*, volume 202 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 53:1–53:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.MFCS.2021.53.

15 Michel Henri Theódore Hack. Petri net language, 1976.

16 Thomas A. Henzinger, Karoliina Lehtinen, and Patrick Totzke. History-deterministic timed automata. In *International Conference on Concurrency Theory*, 2022.

17 Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In *EACSL Annual Conference on Computer Science Logic*, pages 395–410. Springer Berlin Heidelberg, 2006.

18 Piotr Hofman, Slawomir Lasota, Richard Mayr, and Patrick Totzke. Simulation problems over one-counter nets. *Logical Methods in Computer Science*, 12(1), 2016. doi:10.2168/LMCS-12(1:6)2016.

19 Piotr Hofman and Patrick Totzke. Trace inclusion for one-counter nets revisited. *Theoretical Computer Science*, 11174, 2017. doi:10.1016/j.tcs.2017.05.009.

20 Petr Jancar. Nonprimitive recursive complexity and undecidability for Petri net equivalences. *Theoretical Computer Science*, 256(1-2):23–30, 2001. doi:10.1016/S0304-3975(00)00100-6.

21 Petr Jancar, Javier Esparza, and Faron Moller. Petri nets and regular processes. *Journal of Computer and System Sciences*, 59(3):476–503, 1999. doi:10.1006/jcss.1999.1643.

22 Matthias Jantzen. Language theory of Petri nets. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 254 of *LNCS*, pages 397–412. Springer, 1986. doi:10.1007/BFb0046847.

23 Petr Jančar and Faron Moller. Checking regular properties of Petri nets. In *International Conference on Concurrency Theory*, pages 348–362, 1995.

24 Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi:10.1016/S0022-0000(69)80011-5.

25 S. Rao Kosaraju. Decidability of reachability in vector addition systems. In *STOC'82*, pages 267–281, 1982. doi:10.1145/800070.802201.

26 Orna Kupferman, Shmuel Safra, and Moshe Y Vardi. Relating word and tree automata. *Annals of Pure and Applied Logic*, 138(1-3):126–146, 2006.

27 Karoliina Lehtinen and Martin Zimmermann. Good-for-games ω-pushdown automata. *Logical Methods in Computer Science*, 18, 2022.

28 Jérôme Leroux. The reachability problem for Petri nets is not primitive recursive. In *Annual Symposium on Foundations of Computer Science*, pages 1241–1252. IEEE, 2021. doi:10.1109/FOCS52979.2021.00121.

29 Richard J. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.

30 Marvin L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Inc., 1967.

31 Reino Niskanen, Igor Potapov, and Julien Reichert. Undecidability of two-dimensional robot games. In *International Symposium on Mathematical Foundations of Computer Science*, volume 58 of *LIPIcs*, pages 73:1–73:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.73.

32 Aditya Prakash and K. S. Thejaswini. On history-deterministic one-counter nets. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2023. doi:10.1007/978-3-031-30829-1_11.

33 Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, pages 223–231, 1978. doi:10.1016/0304-3975(78)90036-1.

34 Klaus Reinhardt. Reachability in Petri nets with inhibitor arcs. *Electronic Notes in Theoretical Computer Science*, 223:239–264, 2008. doi:10.1016/j.entcs.2008.12.042.

35 Michal Skrzypczak. Büchi VASS recognise w-languages that are Sigmaˆ1_1 – complete. *CoRR*, abs/1708.09658, 2017. arXiv:1708.09658.

36 Leslie G. Valiant and Mike Paterson. Deterministic one-counter automata. *Journal of Computer and System Sciences*, 10(3):340–350, 1975. doi:10.1016/S0022-0000(75)80005-5.

37 Rüdiger Valk and Guy Vidal-Naquet. Petri nets and regular languages. *Journal of Computer and System Sciences*, 23(3):299–325, 1981. doi:10.1016/0022-0000(81)90067-2.

Games with Trading of Control

Orna Kupferman ✉
School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel

Noam Shenwald ✉
School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel

Abstract

The interaction among components in a system is traditionally modeled by a game. In the turned-based setting, the players in the game jointly move a token along the game graph, with each player deciding where to move the token in vertices she controls. The objectives of the players are modeled by ω-regular winning conditions, and players whose objectives are satisfied get rewards. Thus, the game is non-zero-sum, and we are interested in its stable outcomes. In particular, in the rational-synthesis problem, we seek a strategy for the system player that guarantees the satisfaction of the system's objective in all rational environments. In this paper, we study an extension of the traditional setting by *trading of control*. In our game, the players may pay each other in exchange for directing the token also in vertices they do not control. The utility of each player then combines the reward for the satisfaction of her objective and the profit from the trading. The setting combines challenges from ω-regular graph games with challenges in pricing, bidding, and auctions in classical game theory. We study the theoretical properties of *parity trading games*: best-response dynamics, existence and search for Nash equilibria, and measures for equilibrium inefficiency. We also study the rational-synthesis problem and analyze its tight complexity in various settings.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Logic and verification

Keywords and phrases Parity Games, Rational Synthesis, Game Theory, Auctions

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.19

Related Version *Full Version*: `https://www.cs.huji.ac.il/~ornak/publications/concur23.pdf`

Funding Research is funded by the Israel Science Foundation, Grant 2357/19, and the European Research Council, Advanced Grant ADVANSYNT.

1 Introduction

Synthesis is the automated construction of a system from its specification. A useful way to approach synthesis of *reactive* systems is to consider the situation as a *game* between the system and its environment. Together, they generate a computation, and the system wins if the computation satisfies the specification. Thus, synthesis is reduced to generation of a winning strategy for the system in the game – a strategy that ensures that the system wins against all environments [1, 35].

Nowadays systems have rich structures. More and more systems lack a centralized authority and involve selfish users, giving rise to an extensive study of *multi-agent systems* [2] in which the agents have their own objectives, and thus correspond to *non-zero-sum games* [33]: the outcome of the game may satisfy the objectives of a subset of the agents.

The rich settings in which synthesis is applied have led to more involved definitions of the problem. First, in *rational synthesis* [26, 28, 24, 25, 30], the goal is to construct a system that satisfies the specification in all rational environments, namely environments that are composed of components that have their own objectives and act to achieve their objectives. The system can capitalize on the rationality of the environment, leading to synthesis of specifications that cannot be synthesized in hostile environments. Then, in *quantitative synthesis*, the satisfaction value of a specification in a computation need not be Boolean. Thus, beyond correctness, specifications may describe *quality*, enabling the specifier to prioritize different satisfaction scenarios. For example, the value of a computation may be a value in $\mathbb{N}$, reflecting costs and rewards to events along the computation. A synthesis algorithm aims to construct systems that satisfy their objectives in the highest possible value [3, 5, 6, 18, 20]. *Quantitative rational synthesis* then combines the two extensions, with systems composed of rational components having quantitative objectives [26, 28, 6, 19].

Viewing synthesis as a game has led to a fruitful exchange of ideas between *formal methods* and *game theory* [17, 27]. The extensions to rational and quantitative synthesis make the connection between the two communities stronger. Indeed, rationality is a prominent notion in game theory, and most studies in game theory involve quantitative utilities for the players. Classical game theory concerns games for economy-driven applications like resource allocation, pricing, bidding, auctions, and more [37, 33]. Many more useful ideas in classical game theory are waiting to be explored and used in the context of synthesis [23]. In this paper, we introduce and study a framework for extending synthesis with *trading of control*. For example, in a communication network in which each company controls a subset of the routers, companies may pay each other in exchange for committing on some routing decisions, and in a system consisting of a server and clients, clients may pay the server for allocating resources in some beneficial way. The decisions of the players in such settings depend on both their behavioral objectives and their desire to maximize the profit from the trade. When a media company decides, for example, how many and which advertisements it broadcasts, its decisions depend not only on the expected revenue but also on its need to limit the volume (and hopefully also content) of commercial content it broadcasts [16, 31]. More examples include *shields* in synthesis, which can alter commands issued by a controller, aiming to guarantee maximal performance with minimal interference [7, 9].

Our framework considers multi-agent systems modeled by a game played on a graph. Since we care about infinite on-going behaviors of the system, we consider infinite paths in the graph, which correspond to computations of the system. We study settings in which each of the players has control in different parts of the system. Formally, if there are n players, then there is a partition $V_1, \ldots, V_n$ of the set of vertices in the game graph among the players, with Player i controlling the vertices in V_i. The game is *turn-based*: starting from an initial vertex, the players jointly move a token along the game graph, with each player deciding where to move the token in vertices she controls. A *strategy* for Player i directs her how to move a token that reaches a vertex in V_i. A *profile* is a vector of strategies, one for each player, and the *outcome* of a profile is the path generated when the players follow their strategies in the profile. The objectives of the players refer to the generated path. In classical *parity games* (PGs, for short), they are given by *parity* winning conditions over the set of vertices of the graph. Thus, each player has a coloring that assigns numbers to vertices in the graph, and her objective is that the minimal color the path visits infinitely often is even. While satisfaction of the parity winning condition is Boolean, the players get quantitative rewards for satisfying their objectives.

In *parity trading games* (PTG, for short), a strategy for Player i is composed of two strategies: a *buying strategy*, which specifies, for each edge $\langle v, u\rangle$ in the game, how much Player i offers to pay the player that controls v in exchange for this player selling $\langle v, u\rangle$; that is, for always choosing u as v's successor; and a *selling strategy*, which specifies, for each vertex $v \in V_i$, which edge from v is sold, as a function of the offers that Player i receives from the other players. Note that Player i need not sell the edge that gets the highest offer. Indeed, her choice also depends on her objective. Also note that selling strategies are similar to memoryless strategies in PGs, in the sense that a sold edge is going to be traversed in all the visits of the token to its source vertex, regardless of the history of the path. Recall that we consider parity winning conditions, which admits memoryless winning strategies. Accordingly, if a player can force the satisfaction of her parity objective in a PG she can also force the satisfaction of her parity objective in the corresponding PTG.

A profile of strategies in a PTG induces a set of sold edges, one from each vertex. Hence, as in PGs, the outcome of each profile is a path in the game. The utility of Player i in the game is the sum of two factors: a *satisfaction profit*, which, as in PGs, is a reward that Player i receives if the outcome satisfies her objective, and a *trading profit*, which is the sum of payments she receives from the other players, minus the sum of payments she gives others, where payments are made only for sold edges.

Related work studies synthesis of systems that combine behavioral and monetary objectives. One direction of work considers systems with *budgets*. The budget can be used for tasks such as sensing of input signals, purchase of library components [22, 15, 4], and, in the context of control – shielding a controller that interacts with a plant [7, 9]. Even closer is work in which the players can use the budget in order to negotiate control. The most relevant work here is on *bidding games* [12]: graph games in which in each turn an auction is held in order to determine which player gets control. That is, whenever the token is on a vertex v, the players submit bids, the player with the highest bid wins, she decides to which successor of v to move the token, and the budgets of the players are updated according to the bids. Variants of the game refer to its duration, the type of objectives, the way the budgets are updated, and more [13, 14, 11]. Trading games are very different from bidding games: in trading games, negotiation about buying and selling of control takes place before the game starts, and no auctions are held during the game. Also, the games include an initial partition of control, as is the natural setting in multi-agent systems. Moreover, control in trading games is not sold to the highest offer. Rather, selling strategies may depend in the objective of the seller. Finally, the games are non-zero-sum, and are studied for arbitrary number of players.

Another direction of related work considers systems with dynamic change of control that do not involve monetary objectives, such as *pawn games* [10]: zero-sum turn-based games in which the vertices are statically partitioned between a set of *pawns*, the pawns are dynamically partitioned between the players, and the player that chooses the successor for a vertex v at a given turn is the player that controls the pawn to which v belongs. At the end of each turn, the partition of the pawns among the players is updated according to a predetermined mechanism.

Since a PTG is non-zero-sum, interesting questions about it concern *stable outcomes*, in particular *Nash equilibria* (NE) [32]. A profile is an NE if no player has a beneficial deviation; thus, no player can increase her utility by changing her strategy in the profile. Note that in PTGs, a change of a strategy amounts to a change in the buying or selling strategies, or in both of them.

We first study *best response* in PTGs – the problem of finding the most beneficial deviation for a player in a given profile. We show that the problem can be reduced to the problem of finding shortest paths in weighted graphs. Essentially, the weights in the graph are induced by the maximal profit that a player can make from selling edges from vertices she owns and the minimal profit she may lose in order to buy edges from vertices she does not own. We conclude that the problem can be solved in polynomial time. We also study *best response dynamics* – a process in which, as long as the profile is not an NE, some player is chosen to perform her best response. We show that trading makes the setting less stable, in the sense that best response dynamics need not converge to an NE, even when convergence is guaranteed in the underlying PG. On the positive side, as is the case in PGs, every PTG has an NE.

We continue and study rational synthesis in PTGs. Two approaches to rational synthesis have been studied. In *cooperative* rational synthesis (CRS) [26], the desired output is an NE profile whose outcome satisfies the objective of the system. In *non-cooperative* rational synthesis (NRS) [28], we seek a strategy for the system such that its objective is satisfied in the outcome of all NE profiles that include this strategy. In settings with quantitative utilities, in particular PTGs, the input to the CRS and NRS problems includes a threshold $t \geq 0$, and we replace the requirement for the system to satisfy her objective by the requirement that her utility is at least t. The two approaches have to do with the technical ability to communicate strategies to the environment players, say due to different architectures, as well as with the willingness of the environment players to follow a suggested strategy. As shown in [6], the two approaches are related to the two stability-inefficiency measures of *price of stability* (PoS) [8] and *price of anarchy* (PoA) [29, 34], and we study these measures in the context of PTG.

Problem	Finding an NE	Cooperative Rational Synthesis	Non-cooperative Rational Synthesis
Parity Games	UP $\cap$ co-UP fixed n NP-complete unfixed n [37], [Th. 5]	UP $\cap$ co-UP fixed n NP-complete unfixed n [22], [37]	PSPACE, NP-hard, co-NP-hard fixed n EXPTIME, PSPACE-hard unfixed n [22]
Parity Trading Games		NP-complete [Th. 10]	NP-complete $n = 2$ Σ_2^{P}-complete $n \geq 3$ [Th. 12], [Th. 13]
Büchi Games	PTIME [37], [Th. 5]	PTIME [37]	PTIME fixed n PSPACE-complete unfixed n [22]
Büchi Trading Games		NP-complete [Th. 10]	NP-complete $n = 2$ Σ_2^{P}-complete $n \geq 3$ or unfixed n [Th. 12], [Th. 13]

Figure 1 Complexity of different problems on n-player PGs, PTGs, BGs, and BTGs.

In PGs, the tight complexity of rational synthesis is still open, and depends on whether the number of players is fixed. We show that in PTGs, CRS is NP-complete, and the complexity of NRS depends on the number of players: it is NP-complete for two players and is Σ_2^{P}-complete for three or more (in particular, unfixed number of) players. Our upper bounds are based on reductions to a sequence of shortest-path algorithms in weighted graphs. They hold also for an unfixed number of players, making rational synthesis with an unfixed number of players easier in PTGs than in PGs. Intuitively, it follows from the fact that deviations in the selling or buying strategies of single players in PTGs induce a change in the outcome only if they are matched by the buying and selling strategies, respectively, of players that do not deviate. Our lower bounds involve reductions from SAT and QBF$_2$, where trade is used to incentive a satisfying assignment, when exists, and to ensure the consistency of suggested assignments. When the number of players in the environment is bigger than 2, we can use trade among the environment players in order to simulate universal quantification, which explains the transition form NP to Σ_2^{P}.

Our complexity results on ω-regular trading games and their comparison to standard ω-regular non-zero-sum games are summarized in the table in Figure 1. Due to the lack of space, examples and some proofs are omitted or given partially, and can be found at the full version.

2 Preliminaries

For $n \geq 1$, let $[n] = \{1, \ldots, n\}$. An *n-player game graph* is a tuple $G = \langle \{V_i\}_{i \in [n]}, v_0, E \rangle$, where $\{V_i\}_{i \in [n]}$ are disjoint sets of vertices, each owned by a different player, and we let $V = \bigcup_{i \in [n]} V_i$. Then, $v_0 \in V_1$ is an initial vertex, which we assume to be owned by Player 1, and $E \subseteq V \times V$ is a total edge relation, thus for every $v \in V$, there is at least one $u \in V$ such that $\langle v, u \rangle \in E$. The *size* $|G|$ of G is $|E|$, namely the number of edges in it.

For every vertex $v \in V$, we denote by $\mathsf{succ}(v)$ the set of successors of v in G. That is, $\mathsf{succ}(v) = \{u \in V : \langle v, u \rangle \in E\}$. Also, for every $v \in V$, we denote by E_v the set of edges from v. That is, $E_v = \{\langle v, u \rangle : u \in \mathsf{succ}(v)\}$. Then, for every $i \in [n]$, we denote by E_i the set of edges whose source vertex is owned by Player i. That is, $E_i = \bigcup_{v \in V_i} E_v$.

In the beginning of the game, a token is placed on v_0. The players control the movement of the token in vertices they own: In each turn in the game, the player that owns the vertex with the token chooses a successor vertex and moves the token to it. Together, the players generate a *play* $\rho = v_0, v_1, \ldots$ in G, namely an infinite path that starts in v_0 and respects E: for all $i \geq 0$, we have that $(v_i, v_{i+1}) \in E$.

For a play $\rho = v_0, v_1, \ldots$, we denote by $\mathit{inf}(\rho)$ the set of vertices visited infinitely often along ρ. That is, $\mathit{inf}(\rho) = \{v \in V :$ there are infinitely many $i \geq 0$ such that $v_i = v\}$. A *parity* objective is given by a coloring function $\alpha : V \to \{0, \ldots, k\}$, for some $k \geq 0$, and requires the minimal color visited infinitely often along ρ to be even. Formally, a play ρ satisfies α iff $\min\{\alpha(v) : v \in \mathit{inf}(\rho)\}$ is even. A *Büchi* objective is a special case of parity. For simplicity, we describe a Büchi objective by a set of vertices $\alpha \subseteq V$. The condition requires that some vertex in α is visited infinitely often along ρ, thus $\mathit{inf}(\rho) \cap \alpha \neq \emptyset$.

A *parity game* (PG, for short) is a tuple $\mathcal{G} = \langle G, \{\alpha_i\}_{i \in [n]}, \{R_i\}_{i \in [n]} \rangle$, where G is a n-player game graph, and for every $i \in [n]$, we have that $\alpha_i : V \to \{0, \ldots, k_i\}$ is a parity objective for Player i. Intuitively, for every $i \in [n]$, Player i aims for a play ρ that satisfies her objective α_i, and $R_i \in \mathbb{N}$ is a reward that Player i gets when α_i is satisfied. Büchi games (BG, for short) are defined similarly, with Büchi objectives. We assume that at least one condition is satisfiable.

A *strategy* for Player i is a function $f_i : V^* \cdot V_i \to V$ that directs her how to move the token in vertices she owns. Thus, f_i maps prefixes of plays to possible extensions in a way that respects E: for every $\rho \cdot v$ with $\rho \in V^*$ and $v \in V_i$, we have that $(v, f_i(\rho \cdot v)) \in E$. A strategy f_i for Player i is *memoryless* if it only depends on the current vertex. That is, if for every two histories $h, h' \in V^*$ and vertex $v \in V_i$, we have that $f_i(h \cdot v) = f_i(h' \cdot v)$. Note that a memoryless strategy can be viewed as a function $f_i : V_i \to V$.

A *profile* is a tuple $\pi = \langle f_1, \ldots, f_n \rangle$ of strategies, one for each player. The *outcome* of a profile $\pi = \langle f_1, \ldots, f_n \rangle$ is the play obtained when the players follow their strategies. Formally, $\mathsf{Outcome}(\pi) = v_0, v_1, \ldots$ is such that for all $j \geq 0$, we have that $v_{j+1} = f_i(v_0, v_1, \ldots, v_j)$, where $i \in [n]$ is such that $v_j \in V_i$. For every profile π and $i \in [n]$, we say that Player i *wins in* π if $\mathsf{Outcome}(\pi) \models \alpha_i$. Otherwise, Player i *loses in* π. We denote by $\mathsf{Win}(\pi)$ the set of players that win in π. Then, the *satisfaction profit of Player* i *in* π, denoted $\mathsf{sprofit}_i(\pi)$, is R_i if $i \in \mathsf{Win}(\pi)$, and is 0 otherwise.

As the objectives of the players may overlap, the game is not zero-sum and thus we are interested in *stable* profiles in the game. A profile $\pi = \langle f_1, \ldots, f_n \rangle$ is a *Nash Equilibrium* (NE, for short) [32] if, intuitively, no player can benefit (that is, increase her profit) from unilaterally changing her strategy. Formally, for $i \in [n]$ and some strategy f'_i for Player i, let $\pi[i \leftarrow f'_i] = \langle f_1, \ldots, f_{i-1}, f'_i, f_{i+1}, \ldots, f_n \rangle$ be the profile in which Player i *deviates* to the strategy f'_i. We say that π is an NE if for every $i \in [n]$, we have that $\mathsf{sprofit}_i(\pi) \geq \mathsf{sprofit}_i(\pi[i \leftarrow f'_i])$, for every strategy f'_i for Player i. That is, no player can unilaterally increase her profit.

In *rational synthesis*, we consider a game between a system, modeled by Player 1, and an environment composed of several components, modeled by Players $2 \ldots n$. Then, we seek a strategy for Player 1 with which she wins, assuming rationality of the other players. Note that the system may also be composed of several components, each with its own objective. It is not hard to see, however, that they can be merged to a single player whose objective is the conjunction of the underlying components.

We say that a profile $\pi = \langle f_1, \ldots, f_n \rangle$ is a 1-*fixed NE*, if no player $i \in [n] \setminus \{1\}$ has a beneficial deviation. We formalize the intuition behind rational synthesis in two ways, as follows. Consider an n-player game $\mathcal{G} = \langle G, \{\alpha_i\}_{i\in[n]}, \{R_i\}_{i\in[n]}\rangle$, and a threshold $t \geq 0$. The problem of *cooperative rational synthesis* (CRS) is to return a 1-fixed NE π such that $\mathsf{sprofit}_1(\pi) \geq t$. The problem of *non-cooperative rational synthesis* (NRS) is to return a strategy f_1 for Player 1 such that for every 1-fixed NE π that extends f_1, we have that $\mathsf{sprofit}_1(\pi) \geq t$.

As in traditional synthesis, one can also define the corresponding decision problems, of *rational realizability*, where we only need to decide whether the desired strategies exist. In order to avoid additional notations, we refer to CRS and NRS also as decision problems.

3 Parity Trading Games

Parity trading games (PTG, for short, or BTG, when the objectives of the players are Büchi objectives) are similar to parity games, except that now, the movement of the token along the game graph depends on trade among the players, who pay each other in exchange for certain behaviors. Thus, instead of strategies that direct them how to move the token, now the players have strategies that direct the trade.

Consider a PTG $\mathcal{G} = \langle G, \{\alpha_i\}_{i\in[n]}, \{R_i\}_{i\in[n]}\rangle$, defined on top of a game graph $G = \langle \{V_i\}_{i\in[n]}, v_0, E\rangle$. A *buying strategy* for Player i is a function $b_i : E \to \mathbb{N}$ that maps each edge $e = \langle v, u\rangle \in E$ to the price that Player i is willing to pay to the owner of v in exchange for selling e; that is, for always choosing u as v's successor when the token is in v. For edges $e \in E_i$, we require $b_i(e)$ to be 0.

Consider a vector $\beta = \langle b_1, \ldots, b_n\rangle$ of buying strategies, one for each player. The vector β determines, for an edge $e \in E$, the collective price that the players are willing to pay for e. Accordingly, we sometime refer to β as a *price list*, namely a function in $\mathbb{N}^E$, where for every $e \in E$, we have that $\beta(e) = \sum_{i\in[n]} b_i(e)$.

A *selling strategy* for Player i determines which edges Player i sells. The strategy is a collection of policies, which determines for each $v \in V_i$, which edge from v to sell, given prices offered for the edges in E_v. Formally, a *selling policy for* $v \in V_i$ is a function $s_v : \mathbb{N}^{E_v} \to E_v$ that maps each price list for the edges in E_v to an edge in E_v. Note that the mapping is arbitrary, thus a player need not sell the edge that gets the highest price. We refer to the selling strategy for Player i, thus the collection $\{s_v : v \in V_i\}$ of selling policies for her vertices, as a function $s_i : \mathbb{N}^E \to 2^{E_i}$ that maps price lists to the set of edges that Player i chooses to sell. Note also that selling strategies in PTGs are similar to memoryless strategies in PGs, in the sense that the choice of the edge that is sold from v is independent of the history of the game.

A *profile* is a tuple $\pi = \langle (b_1, s_1), \ldots, (b_n, s_n)\rangle$ of pairs of buying and selling strategies, one for each player. We sometime refer to the pair of buying and selling strategies for Player i as a single strategy, and use the notation $f_i = (b_i, s_i)$. We also use β_π to denote the price list induced by the buying strategies in π. We say that an edge $e \in E_i$ is *sold* in π iff $e \in s_i(\beta_\pi)$. We denote by $\mathsf{S}(\pi)$ the set of edges sold in π. Recall that for every $v \in V$, there exists exactly one edge $e \in E_v$ such that $e \in \mathsf{S}(\pi)$. The *outcome* of a profile π, denoted $\mathsf{Outcome}(\pi)$, is then the path $v_0, v_1, \ldots$, where for all $j \geq 0$, we have that $(v_j, v_{j+1}) \in \mathsf{S}(\pi)$.

As in PGs, the satisfaction profit of Player i in π, denoted $\mathsf{sprofit}_i(\pi)$, is R_i if α_i is satisfied in $\mathsf{Outcome}(\pi)$, and is 0 otherwise. In PTGs, however, we consider also the trading profits of the players: For every player $i \in [n]$, the *gain* of Player i in π, denoted $\mathsf{gain}_i(\pi)$, is the sum of payments she receives from other players, and the *loss* of Player i, denoted

$\mathsf{loss}_i(\pi)$, is the sum of payments she pays others. That is, $\mathsf{gain}_i(\pi) = \sum_{e \in \mathsf{S}(\pi) \cap E_i} \beta_\pi(e)$, and $\mathsf{loss}_i(\pi) = \sum_{e \in \mathsf{S}(\pi)} b_i(e)$. Then, the *trading profit* of Player i in π, denoted $\mathsf{tprofit}_i(\pi)$, is her gain minus her loss in π. That is, $\mathsf{tprofit}_i(\pi) = \mathsf{gain}_i(\pi) - \mathsf{loss}_i(\pi)$. Note that while all the edges in $\mathsf{Outcome}(\pi)$ are in $\mathsf{S}(\pi)$, not all edges in $\mathsf{S}(\pi)$ are traversed during the play. Still, payments depend only on $\mathsf{S}(\pi)$, regardless of whether the edges are traversed. Finally, the *utility* of Player i in π, denoted $\mathsf{util}_i(\pi)$, is the sum of her satisfaction and trading profits in π. That is, $\mathsf{util}_i(\pi) = \mathsf{sprofit}_i(\pi) + \mathsf{tprofit}_i(\pi)$. The definitions of beneficial deviations, NEs, and 1-fixed NEs are then defined as in the case of PG.

Note that the definition of a selling strategy s_i as a function from $\mathbb{N}^E$ hides the fact that the selling policy for each vertex $v \in V_i$ depends only on the price list for the edges in E_v. Note also that as there are infinitely many price lists, an enumerative presentation of selling strategies is infinite. As we detail in the full version, we assume that selling strategies are given symbolically. For example, a selling strategy for a vertex v with successors $\{u_1, u_2, u_3\}$, may be "if the price offered for u_2 is at least p, then sell (v, u_2); otherwise, sell (v, u_1)". Specifically, a strategy for Player i is given by a set of pairs of the form $\langle b, T \rangle$, where b is a predicate on $\mathbb{N}^E$ and $T \subseteq E_i$ is the set of edges that Player i sells when then price list satisfies b. The predicates are disjoint, and can be computed in polynomial time. In the full version we also argue that every profile π of strategies can be simplified so that the set of winners and the utilities for the players are preserved, and all prices are of polynomial size. As we argue in the sequel, restricting attention to simple profiles and to strategies that can be represented symbolically does not lose generality, in the sense that whenever we search for a profile of strategies and a desired profile exists, then there is also a profile that consists of strategies that can be represented symbolically.

Describing a profile $\pi = \langle (b_1, s_1), \ldots, (b_n, s_n) \rangle$, we sometimes use a symbolic description, as follows. For players $i, j \in [n]$, an edge $e \in E_j$, and a price $p \in \mathbb{N}$, we say that Player i *offers to buy e for price p* if $b_i(e) = p$, and that Player i *pays p for e* if, in addition, $e \in s_j(\beta_\pi)$. For a vertex $v \in V_i$, and an edge $e = \langle v, u \rangle \in E_v$, we say that Player i *moves from v to u*, if $e \in s_i(\beta_\pi)$, thus Player i sells e in β_π. Then, we say that Player i *always moves from v to u*, if Player i always sells e, thus $e \in s_i(\beta)$ for every price list β. Describing a deviation from π to a profile $\pi' = \langle (b'_1, s'_1), \ldots, (b'_n, s'_n) \rangle$, we sometimes use a symbolic description, as follows. For a player $i \in [n]$ and an edge $e \in E$, we say that Player i *cancels the purchase of e* if $b_i(e) > 0$ and $b'_i(e) = 0$. For an edge $e \in E_i$, we say that Player i *cancels the sale of e* if $e \in s_i(\beta_\pi)$ and $e \notin s_i(\beta_{\pi'})$.

4 Stability in Parity Trading Games

In this section we study the stability of PTGs. We start with the best-response problem, which searches for deviations that are most beneficial for the players, and show that the problem can be solved in polynomial time. On the negative side, a best-response dynamics in PTGs, where players repeatedly perform their most beneficial deviations, need not converge. We then study the existence of NEs in PTGs, show that every PTG has an NE, and relate the stability in a PTG and its underlying PG. Finally, we study the inefficiency that may be caused by instability, and show that the price of stability and price of anarchy in PTGs are unbounded and infinite, respectively.

Throughout this section, we consider an n-player game $\mathcal{G} = \langle G, \{\alpha_i\}_{i \in [n]}, \{R_i\}_{i \in [n]} \rangle$, defined on top of a game graph $G = \langle \{V_i\}_{i \in [n]}, v_0, E \rangle$. We use $\mathcal{G}^P$ and $\mathcal{G}^T$ to denote $\mathcal{G}$ when viewed as a PG and PTG, respectively.

4.1 Best response

The input to the *best response* (BR, for short) problem is a game $\mathcal{G}$, a profile π, and $i \in [n]$. The goal is to find a strategy f'_i for Player i such that $\mathsf{util}_i(\pi[i \leftarrow f'_i])$ is maximal. We describe an algorithm that solves the BR problem in polynomial time. The key idea behind our algorithm is as follows. Consider a profile $\pi = \langle (b_1, s_1), \ldots, (b_n, s_n) \rangle$. Recall that the utility of Player i in π is the sum of her satisfaction and trading profits in π. If Player i ignores her objective and only tries to maximize her trading profit, then her strategy is straightforward: she buys no edge, and in each vertex $v \in V_i$, she sells an edge with the maximal price in β_π. If there is a strategy f^*_i as above such that the outcome of $\pi[i \leftarrow f^*_i]$ satisfies α_i, then clearly f^*_i is a best response for Player i, and we are done. Otherwise, the algorithm searches for a minimal reduction in the trading profit with which Player i can induce an outcome that satisfies α_i. For this, the algorithm labels each edge $e = \langle v, u \rangle$ in G by the cost of ensuring that e is sold. If Player i owns e, then this cost is the difference between $\beta_\pi(e)$ and $\max\{\beta_\pi(e') : e' \in E_v\}$. If Player i does not own e, thus $v \in V_j$, for some player $j \neq i$, then this cost is the minimal price that Player i has to offer for e in order to change β_π to a price list β for which $s_j(\beta) = e$. Once the graph G is labeled by costs as above, the desired strategy is induced by the path with the minimal cost that satisfies α_i. Finally, if the minimal cost of satisfying α_i is higher than her reward R_i, then the best response for Player i is to give up the satisfaction of α_i and follow the strategy f^*_i, in which the maximal trading profit is attained.

We now describe the algorithm in detail. We first label the edges from every vertex $v \in V$ by costs in $\mathbb{N}$. For every vertex $v \in V_i$, we denote by $\mathsf{potential}(\pi, v)$ the maximal price that Player i can get from selling an edge from v. That is, $\mathsf{potential}(\pi, v) = \max\{\beta_\pi(e) : e \in E_v\}$. For every vertex $v \in V_i$ and edge $e \in E_v$, we define $\mathsf{cost}(\pi, e)$ as the cost for Player i of selling e rather then an edge that attains $\mathsf{potential}(\pi, v)$. That is, $\mathsf{cost}(\pi, e) = \mathsf{potential}(\pi, v) - \beta_\pi(e)$.

We continue to vertices $v \notin V_i$. For $j \in [n] \setminus \{i\}$ and an edge $e \in E_j$, we define $\mathsf{cost}(\pi, e)$ as the minimal price that Player i needs to pay to Player j in order for her to sell e. Formally, let B^e_i be the set of buying strategies for Player i that cause Player j to sell e. That is, $B^e_i = \{b'_i : E \to \mathbb{N} : e \in s_j(\beta_\pi[i \leftarrow b'_i])\}$. When Player i uses a strategy $b'_i \in B^e_i$ as her buying strategy, Player j sells e, and Player i pays the price $b'_i(e)$. Hence, the minimal price that Player i needs to pay in order for Player j to sell e is $\mathsf{cost}(\pi, e) = \min\{b'_i(e) : b'_i \in B^e_i\}$. Note that B^e_i may be empty, in which case $\mathsf{cost}(\pi, e) = \infty$.

We define $\mathsf{best}(\pi) \subseteq E$ as the set of edges that minimize the cost of Player i. Formally, $\mathsf{best}(\pi) = \bigcup_{v \in V} \mathsf{best}(\pi, v)$, where for $v \in V_i$, we have that $\mathsf{best}(\pi, v) \subseteq E_v$ is the set of edges from v with which $\mathsf{potential}(\pi, v)$ is attained, thus $\mathsf{best}(\pi, v) = \{e \in E_v : \beta_\pi(e) = \mathsf{potential}(\pi, v)\}$; and for $v \in V_j$, for $j \neq i$, we have that $\mathsf{best}(\pi, v)$ is the set of edges from v that Player i can make Player j sell without paying for e, thus $\mathsf{best}(\pi, v) = \{e \in E_v : \mathsf{cost}(\pi, e) = 0\}$. Note that for every vertex $v \in V$, the set $\mathsf{best}(\pi, v)$ is not empty.

We say that a path ρ in G is *feasible* if $\mathsf{cost}(\pi, e) < \infty$ for every edge e in ρ. In Lemma 1 below, we argue that for every feasible path ρ, Player i can change her strategy in π so that the outcome of the new profile is ρ. We also calculate the cost required for Player i to do so.

▶ **Lemma 1.** *Let ρ be a feasible path in $\mathcal{G}$. Then, there exists a strategy f^ρ_i for Player i such that $\mathsf{Outcome}(\pi[i \leftarrow f^\rho_i]) = \rho$, and $\mathsf{tprofit}_i(\pi[i \leftarrow f^\rho_i]) = \sum_{v \in V_i} \mathsf{potential}(\pi, v) - \sum_{e \in \rho} \mathsf{cost}(\pi, e)$. Also, $\mathsf{tprofit}_i(\pi[i \leftarrow f^\rho_i])$ is the maximal trading profit for Player i when she changes her strategy in π to a strategy that causes the outcome to be ρ.*

For a path ρ in G, let f^ρ_i be a strategy for Player i such that the outcome of $\pi[i \leftarrow f^\rho_i]$ is ρ. Note that f^ρ_i can be described symbolically.

Our algorithms for finding beneficial deviations are based on a search for short *lassos* in weighted variants of the graph G. A lasso is a path of the form $\rho_1 \cdot \rho_2^\omega$, for finite paths $\rho_1 \in V^*$ and $\rho_2 \in V^+$. When G is weighted, the length of the lasso is defined as the sum of the weights in the path $\rho_1 \cdot \rho_2$.

▶ **Theorem 2.** *The BR problem in PTGs can be solved in polynomial time.*

Proof. Given an n-player PTG $\mathcal{G}$, a profile π, and $i \in [n]$, the algorithm for finding a BR for Player i proceeds as follows.

1. Let $G^{\mathsf{best}(\pi)} = \langle V, \mathsf{best}(\pi)\rangle$ be the restriction of G to edges in $\mathsf{best}(\pi)$.
2. If there is a path ρ in $G^{\mathsf{best}(\pi)}$ that satisfies α_i, then return f_i^ρ. Otherwise, let f_i^* be a strategy for Player i that induces some lasso in $G^{\mathsf{best}(\pi)}$.
3. Let $G' = \langle V, E, w\rangle$ be the weighted extension of G, where $w : E \to \mathbb{N}$ is such that for every edge $e \in E$, we have that $w(e) = \mathsf{cost}(\pi, e)$.
4. Let ρ be a shortest (with respect to the weights in w) lasso that satisfies α_i.
5. If $w(\rho) \geq R_i$, then return f_i^*, else return f_i^ρ. ◀

Recall that a *best response dynamic* (BRD) is an iterative process in which as long as the profile is not an NE, some player is chosen to perform a best response. In Theorem 3 below, we demonstrate that a BRD in a PTG (in fact, a BTG) need not converge, even in settings in which every BRD in the corresponding PG does converge.

▶ **Theorem 3.** *There is a game $\mathcal{G}$ such that every BRD in the PG $\mathcal{G}^P$ converges to an NE, yet a BRD in $\mathcal{G}^T$ need not converge.*

Proof. Consider the 2-player Büchi game $\mathcal{G} = \langle G, \{\alpha_1, \alpha_2\}, \{1, 3\}\rangle$, where G is described in Figure 2, $\alpha_1 = \{a, c\}$, and $\alpha_2 = \{b, d\}$.

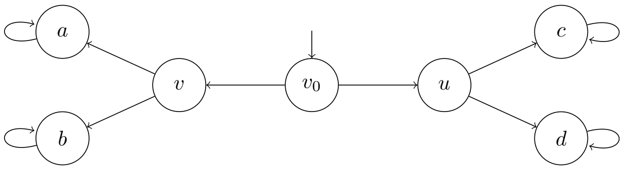


Figure 2 The game graph G. All the vertices are owned by Player 1.

All the vertices in G are owned by Player 1, and the vertices in α_1 are reachable sinks. Hence, once Player 1 is chosen to deviate in $\mathcal{G}^P$, an NE is reached.

In the full version we describe a BRD in $\mathcal{G}^T$ that does not converge. ◀

4.2 Nash equilibria

We continue and show that while a BRD in $\mathcal{G}^T$ need not converge even when every BRD in $\mathcal{G}^P$ does, we can still use NEs in $\mathcal{G}^P$ in order to obtain NEs in $\mathcal{G}^T$. Consider a profile $\pi = \langle f_1, \ldots, f_n\rangle$ of memoryless strategies for the players in $\mathcal{G}^P$. We define the *trivial-trading analogue* of π, denoted $tt(\pi)$ as the a profile in $\mathcal{G}^T$ that is obtained from π by replacing each strategy f_i by the pair (b_i, s_i), for an empty buying strategy b_i (that is, $b_i(e) = 0$ for all $e \in E$), and a selling strategy s_i that mimics f_i (that is, for every price list β, we have that $\langle v, u\rangle \in s_i(\beta)$ iff $f_i(v) = u$). Note that all the strategies in $tt(\pi)$ can be described symbolically.

▶ **Lemma 4.** *If π is an NE in $\mathcal{G}^P$ that consists of memoryless strategies, then $tt(\pi)$ is an NE in $\mathcal{G}^T$.*

Lemma 4 enables us to reduce the search for an NE in an n-player PTG $\mathcal{G}^T$ to a search for an NE in the PG $\mathcal{G}^P$:

▶ **Theorem 5.** *Every PTG has an NE, which can be found in UP ∩ co-UP when the number of players is fixed, and in NP when the number of players is not fixed. For BTGs, an NE can be found in polynomial time.*

Recall that for solving the rational-synthesis problem, we are not interested in arbitrary NEs, but in 1-fixed NEs in which the utility of Player 1 is above some threshold. As we shall see now, the situation here is more complicated: searching for solutions for the rational-synthesis problem in a PTG, we cannot reason about the corresponding PG.

▶ **Theorem 6.** *There is a PTG $\mathcal{G}^T$ and $t \geq 1$ such that there is a 1-fixed NE π^T in $\mathcal{G}^T$ with $\mathsf{util}_1(\pi^T) \geq t$, yet for every 1-fixed NE of memoryless strategies π in $\mathcal{G}^P$, we have that $\mathsf{util}_1(tt(\pi)) < t$.*

Proof. Consider the 2-player BTG $\mathcal{G}^T = \langle G, \{\{a\}, \{b\}\}, \{1, 3\}\rangle$, where G appears in Figure 3. Consider a profile π^T in which the strategy for Player 1 moves from v_0 to b if Player 2 offers to buy $\langle v_0, b\rangle$ for price 2, and moves to a otherwise, and the strategy for Player 2 offers to buy $\langle v_0, b\rangle$ for price 2. In the full version, we prove that π^T is a 1-fixed NE with $\mathsf{util}_1(\pi^T) = 2$, whereas for every 1-fixed NE of memoryless strategies π in $\mathcal{G}^P$, we have that $\mathsf{util}_1(tt(\pi)) < 2$. ◀

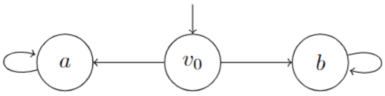


Figure 3 The game graph G. All the vertices are owned by Player 1.

Note that while Theorem 6 considers a 1-fixed NE, and thus corresponds to the setting of CRS, the strategy for Player 1 described there is in fact an NRS solution for the threshold $t = 2$, and the latter cannot be obtained by extending an NRS solution for Player 1 in $\mathcal{G}^P$.

4.3 Equilibrium inefficiency

In this section we study the *price of stability* (PoS) and *price of anarchy* (PoA) measures [33] in PTGs, describing the best-case and worst-case inefficiency of a Nash equilibrium.

Before we define these measures formally, we observe that for every PTG, outcomes that agree on the set of winners also agree in the sum of utilities of the players. Essentially, this follows from the fact that the trading profits for the players sum to 0. Formally, we have the following.

▶ **Lemma 7.** *Let ρ be a path in G, and let $\mathsf{Win}(\rho)$ be the set of players whose objectives are satisfied in ρ. Then, for every profile π with $\mathsf{Outcome}(\pi) = \rho$, we have that the sum of utilities of the players in π is exactly $\sum_{i \in \mathsf{Win}(\rho)} R_i$.*

The *social optimum* in a game $\mathcal{G}$, denoted $\mathsf{SO}(\mathcal{G})$, is the maximal sum of utilities that the players can have in some profile. Thus, $\mathsf{SO}(\mathcal{G})$ is the maximal $\sum_{i \in [n]} \mathsf{util}_i(\pi)$ over all profiles π for $\mathcal{G}$. Since every path ρ in G can be the outcome of some profile, then, by Lemma 7, we have that $\mathsf{SO}(\mathcal{G})$ is the maximal $\sum_{i \in \mathsf{Win}(\rho)} R_i$ over all paths ρ in G.

Let π_B and π_W be NEs with the highest and lowest sum of utilities for the players, respectively. We define $\mathsf{BNE}(\mathcal{G}) = \sum_{i\in[n]} \mathsf{util}_i(\pi_B)$ and $\mathsf{WNE}(\mathcal{G}) = \sum_{i\in[n]} \mathsf{util}_i(\pi_W)$. We then define the price of stability in $\mathcal{G}$ as $\mathsf{PoS}(\mathcal{G}) = \mathsf{SO}(\mathcal{G})/\mathsf{BNE}(\mathcal{G})$, and the price of anarchy in $\mathcal{G}$ as $\mathsf{PoA}(\mathcal{G}) = \mathsf{SO}(\mathcal{G})/\mathsf{WNE}(\mathcal{G})$. Analyzing the prices of stability and anarchy of PTGs, we assume that all rewards in a game $\mathcal{G}$ are positive, thus $R_i > 0$ for all $i \in [n]$. Note that without this assumption, it is easy to define a game $\mathcal{G}$ with $\mathsf{SO}(\mathcal{G}) > 0$ yet $\mathsf{BNE}(\mathcal{G}) = 0$, and hence with $\mathsf{PoS}(\mathcal{G}) = \mathsf{PoA}(\mathcal{G}) = \infty$.

We start with the price of anarchy. It is easy to see that it may be infinite even in simple PTGs in which all rewards are positive:

▶ **Theorem 8.** *There is a 2-player BTG $\mathcal{G}$ with $\mathsf{PoA}(\mathcal{G}) = \infty$.*

Proof. Consider the BTG $\mathcal{G} = \langle G_{PoA}, \{\{a\}, \{a\}\}, \{1, 1\}\rangle$, where the game graph G_{PoA} is described in Figure 4. In the full version we show that $\mathsf{SO}(\mathcal{G}) = 1 + 1 = 2$, whereas $\mathsf{WNE}(\mathcal{G}) = 0$, and so $\mathsf{PoA}(\mathcal{G}) = 2/0 = \infty$. ◀

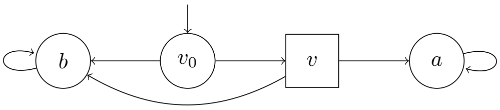


Figure 4 The game graph G_{PoA}. The circles are vertices controlled by Player 1, and the squares are vertices controlled by Player 2.

We continue to the price of stability. It can be shown that every PG has an NE in which all players use memoryless strategies and at least one player satisfies her objective. Essentially, this follows from the fact that either at least one player in the game has a strategy to fulfill her objective from some vertex in all environments (that is, in the zero-sum game played with her objective), or all players do not have such a strategy. In the first case, the outcome of the required NE reaches the winning (in the zero-sum sense) vertex for the player along vertices that are losing (in the zero-sum sense) for the other players. In the second, the outcome traverses a lasso that satisfies the objective of at least one player but consists of vertices that are losing (again, in the zero-sum sense) for all players. By Lemma 4, it then follows that every PTG also has an NE in which at least one player satisfies her objective. Thus, as we assume that all rewards are strictly positive, we conclude that $\mathsf{BNE}(\mathcal{G}) > 0$ for every PTG $\mathcal{G}$. Therefore, we cannot expect $\mathsf{PoS}(\mathcal{G})$ to be ∞, and the strongest result we can prove is that $\mathsf{PoS}(\mathcal{G})$ is unbounded:

▶ **Theorem 9.** *For every $x \in \mathbb{N}$, there exists a two-player BTG $\mathcal{G}$ with $\mathsf{PoS}(\mathcal{G}) = x$.*

Proof. Given x, consider the two-player game graph $G = \langle V_1, V_2, v_1, E\rangle$, where $V_1 = \emptyset$, $V_2 = \{v_1, \ldots, v_{x+2}, u\}$, and $E = \{\langle v_i, v_{i+1}\rangle, \langle v_i, u\rangle : 1 \le i \le x + 1\} \cup \{\langle u, u\rangle, \langle v_{x+2}, v_{x+2}\rangle\}$ (see Figure 5).

Consider the BTG $\mathcal{G} = \langle G, \{\{v_{x+2}\}, \{u\}\}, \{x, 1\}\rangle$. In the full version, we show that $\mathsf{SO}(\mathcal{G}) = x$ whereas $\mathsf{BNE}(\mathcal{G}) = 1$, thus $\mathsf{PoS}(\mathcal{G}) = x$. ◀

5 Cooperative Rational Synthesis in Parity Trading Games

In this section, we study the complexity of the the CRS problem for PTGs and BTGs. Recall that for PGs, the CRS problem can be solved in UP ∩ co-UP when the number of players is fixed, and is in NP when the number of players is not fixed [24]. For BGs, CRS can be solved in polynomial time [36]. We show that trading make the problem harder: CRS in PTGs is NP-complete already for a fixed number of players and for Büchi objectives.

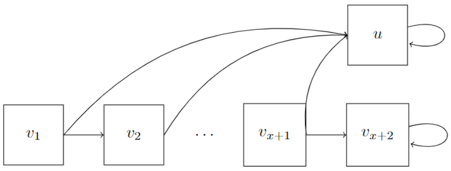


Figure 5 The game graph G. All the vertices are owned by Player 2.

▶ **Theorem 10.** *CRS for PTGs is NP-complete. Hardness in NP holds already for BTGs.*

Proof. We start with membership in NP. Given a threshold $t \geq 0$, an NP algorithm guesses a profile π, checks that $\mathsf{util}_1(\pi) \geq t$, and checks that π is a 1-fixed NE as follows. For every $i \in [n] \setminus \{1\}$, it finds the best response f_i^* for Player i in π, and checks that $\mathsf{util}_i(\pi) \geq \mathsf{util}_i(\pi[i \leftarrow f_i^*])$, thus Player i has no beneficial deviation in π. By Theorem 2, finding the best response for each player in π can be done in polynomial time, hence the check is in polynomial time.

For the lower bound, we describe a reduction from 3-SAT to CRS in BTGs. Let $X = \{x_1, \ldots, x_n\}$, $\overline{X} = \{\overline{x}_1, \ldots, \overline{x}_n\}$, and let φ be a Boolean formula over the variables in X, given in 3CNF. That is, $\varphi = (l_1^1 \vee l_1^2 \vee l_1^3) \wedge \cdots \wedge (l_k^1 \vee l_k^2 \vee l_k^3)$, where for all $1 \leq i \leq k$ and $1 \leq j \leq 3$, we have that $l_i^j \in X \cup \overline{X}$. For every $1 \leq i \leq k$, let $C_i = (l_i^1 \vee l_i^2 \vee l_i^3)$.

Given a formula φ, we construct (see Figure 6) a two-player BG $\mathcal{G} = \langle G_{SAT}, \{\alpha_1, \alpha_2\}, \{R_1, R_2\}\rangle$, where $\alpha_1 = V \setminus \{s\}$, $\alpha_2 = \{s\}$, $R_1 = n + 1$ and $R_2 = 1$, such that φ is satisfiable iff there exists a 1-fixed NE π in $\mathcal{G}$ in which $\mathsf{util}_1(\pi) \geq 1$. The main idea of the reduction is that Player 1 chooses an assignment to the variables in X, and then Player 2 challenges the assignment by choosing a clause of φ. The objective of Player 1 is to not get stuck in a sink, and the objective of Player 2 is to get stuck in the sink. Whenever Player 1 chooses an assignment to a variable, Player 2 has an opportunity to go to the sink, and Player 1 has to buy an edge in order to prevent her from doing so. The reward R_1 for Player 1 is $n+1$, and so Player 1 can buy n edges and still have utility 1. If Player 1 chooses an assignment that satisfies φ, then she can prevent the game from going to the sink by buying only n edges – one for each variable. Otherwise, Player 2 can choose a clause that is not satisfied by the assignment, which forces Player 1 to buy more than n edges or give up the prevention of the sink. ◀

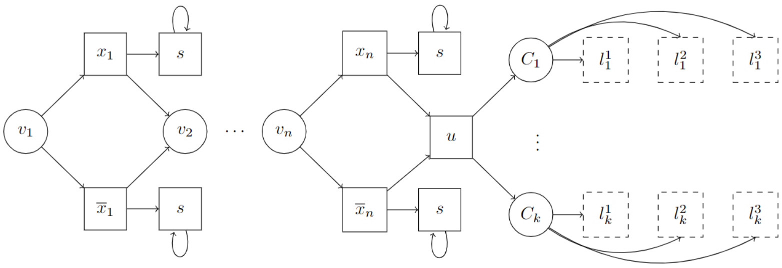


Figure 6 The game graph G_{SAT}. The circles are vertices owned by Player 1, and the squares are vertices owned by Player 2. The dashed vertices are the corresponding literal vertices on the assignment part of the graph.

6 Non-cooperative Rational Synthesis in Parity Trading Games

In this section we study NRS for PTGs. Recall that in PGs, the NRS problem is in PSPACE when the number of players is fixed, and can be solved in exponential time when their number is not fixed [24]. In BGs, NRS can be solved in polynomial time when the number of players is fixed, and the problem is PSPACE-complete when the number of players is not fixed. We show that the NRS problem in PTGs and BTGs is NP-complete for games with two players, and is Σ_2^{P}-complete for games with three or more players.

6.1 Two-player NRS

Consider a game $\mathcal{G} = \langle G, \{\alpha_1, \alpha_2\}, \{R_1, R_2\}\rangle$, a strategy $f_1 = (b_1, s_1)$ for Player 1, and a threshold $t \geq 0$. We describe an algorithm that determines if f_1 is an NRS solution for t in polynomial time. The key idea behind our algorithm is as follows. Let U_2 be the maximal utility for Player 2 in a profile π that extends f_1. Then, as Player 2 can ensure she gets utility of U_2, we have that every profile π in which $\mathsf{util}_2(\pi) = U_2$ is a 1-fixed NE, and every profile π in which $\mathsf{util}_2(\pi) < U_2$ is not a 1-fixed NE. Hence, f_1 is an NRS solution iff for every profile π that extends f_1 with $\mathsf{util}_2(\pi) = U_2$, we have that $\mathsf{util}_1(\pi) \geq t$.

We now describe the algorithm in detail. The algorithm first labels the edges from every vertex $v \in V$ by costs in $\mathbb{N}$. Recall the weights $\mathsf{cost}(\pi, e)$ described in Section 4 in the context of deviations for Player i. Observe that $\mathsf{cost}(\pi, e)$ is independent of the strategy f_i of Player i in π. In particular, when we consider deviations for Player 2, we have that $\mathsf{cost}(\pi, e)$ depends only on the function f_1 of Player 1, and can thus be denoted $\mathsf{cost}(f_1, e)$.

▶ **Lemma 11.** *Checking whether a given strategy for Player 1 is an NRS solution in a PTG can be done in polynomial time.*

Proof. Consider a PTG $\mathcal{G} = \langle G, \{\alpha_1, \alpha_2\}, \{R_1, R_2\}\rangle$, a strategy f_1 for Player 1, and a threshold $t \geq 0$. Let $G = \langle V, E\rangle$.

1. Let $G' = \langle V, E, w\rangle$ be a weighted version of G, where for every edge $e \in E$, we have that $w(e) = \mathsf{cost}(f_1, e)$.
2. For every $W \subseteq \{1, 2\}$, let ρ_W be the shortest lasso in G' such that the set of winners in ρ_W is W. Let f_2^W denote the corresponding strategy for Player 2.
3. Let $U_2 = \max\{\mathsf{util}_2(\langle f_1, f_2^W\rangle) : W \subseteq \{1, 2\}\}$. Note that U_2 is the maximal utility that Player 2 can get when the strategy for Player 1 is f_1.
4. If there exists a set $W \subseteq \{1, 2\}$ such that $\mathsf{util}_2(\langle f_1, f_2^W\rangle) = U_2$ and $\mathsf{util}_1(\langle f_1, f_2^W\rangle) < t$, then f_1 is not a NRS solution. Otherwise, f_1 is an NRS solution. ◀

Lemma 11 implies an NP upper bound for NRS for 2-players PTGs. A matching lower bound is proven by a reduction from 3SAT.

▶ **Theorem 12.** *NRS for 2-players PTGs is NP-complete. Hardness in NP holds already for BTGs.*

6.2 n-player NRS for $n \geq 3$

We continue and study NRS for PTGs with strictly more than two players. As bad news, we show that the polynomial algorithm from the proof of Theorem 12 cannot be generalized for NRS with three or more players. Intuitively, the reason is as follows. In the case of two players, there is a single environment player, and when the strategy for the system player is fixed, we could find the maximal possible utility for the environment player. On the other hand, when there are two or more environment players, the maximal possible utility for each of them depends on both the strategy of the system player and the strategies of the other environment players, which are not fixed. Formally, we prove that NRS for PTGs with strictly more than two players is Σ_2^{P}-complete. As good news, NRS stays Σ_2^{P} also when the number of players in not fixed; thus is is easier than NRS in PGs, where the problem is PSPACE-hard for an unfixed number of players.

▶ **Theorem 13.** *NRS for n-players PTGs with $n \geq 3$ is Σ_2^{P}-complete. Hardness in Σ_2^{P} holds already for BTGs.*

Proof. We start with the upper bound. We say that a profile π is *good* if $\mathsf{util}_1(\pi) \geq t$, or π is not a 1-fixed NE. Checking whether a given profile π is good can be done in polynomial time. Indeed, for checking whether $\mathsf{util}_1(\pi) \geq t$, we can find $\mathsf{S}(\pi)$ and $\mathsf{Outcome}(\pi)$, and then calculate $\mathsf{util}_1(\pi)$ in polynomial time. For checking whether π is not a 1-fixed NE, we can use Theorem 2 and check if some player $i \in [n] \setminus \{1\}$ has a beneficial deviation. Hence, an algorithm in Σ_2^{P} for NRS guesses a strategy f_1 for Player 1 and then checks that for all guessed strategies $f_2, \ldots, f_n$ for Players $2 \ldots n$, the profile $\langle f_1, f_2, \ldots, f_n \rangle$ is good. Note that the complexity is independent of n being fixed.

We continue to the lower bound and show that NRS is Σ_2^{P}-hard already for three players in BTGs. We describe a reduction from QBF$_2$, the problem of determining the truth of quantified Boolean formulas with one alternation of quantifiers, where the external quantifier is "exists". Consider a QBF$_2$ formula $\Phi = \exists x_1, \ldots, x_n \forall y_1, \ldots, y_m \varphi$. We assume that φ is a Boolean propositional formula in 3DNF. Let $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$. Given Φ, we construct a 3-player Büchi game such that there exists an NRS solution f_1 in $\mathcal{G}$ for $t = 1$ iff $\Phi = \mathbf{true}$.

The main idea of the reduction is to construct a game in which Player 1 chooses an assignment to the variables in X; Player 2 tries to prove that $\Phi = \mathbf{false}$, by showing that there exists an assignment to the variables in Y with which for every clause C_i, there is a literal l_i^j such that $l_i^j = \mathbf{false}$; and Player 3 can point out whenever Player 2's proof is incorrect. The game has a sink s. The objective of Player 1 and Player 3 is to not get stuck in the sink, and the objective of Player 2 is V. That is, Player 2 wins in every path in the game. The reward to Player 1 is $n + 1$, and she can pay 1 for each assignment in order to ensure that the play does not reach s. If Player 1 chooses an assignment for the variables in X such that for every assignment to the variables in Y, we have that φ is satisfied, then she and Player 3 can prevent the game from going to s, with Player 1 paying a total price of n. Otherwise, Player 2 can prove that $\Phi = \mathbf{false}$, and by that forces the play to reach s, unless Player 1 pays more than n, which exceeds her reward. ◀

7 Discussion

We introduced trading games, which extend ω-regular graph games with trading of control. Our buying and selling strategies concern edges in the game graph, and the result of the trading is a set of sold edges. In this section we discuss richer settings, classified according to the parameter they extend the setting with.

Buying strategies. We see two interesting ways to enrich buying strategies. The first, which is common in game theory, is to allow *dependencies* between the sold goods, thus let players bid on sets of edges [33]. Indeed, a company may be willing to pay for the rights to direct the traffic in a certain router in a communication network only if it also gets the right to direct traffic in a certain neighbour router. While it is not hard to extend our results to a setting with such dependencies, it makes the description of strategies more complex. The second way concerns the type of control that is traded. Rather than buying edges, a player may buy ownership of vertices. In the case of games with objectives that only require memoryless strategies, the difference boils down to *information*: the new owner is still going to use the same edge in all visits to a vertex she bought, yet unlike in our setting, the seller of the vertex does not known which edge it is. For games in which memoryless strategies are too weak (for example, games with generalized parity objectives, or objectives in LTL [21]), the suggested model allows the buyer to proceed with different edges in different visits to the sold vertex. Moreover, by allowing buying strategies that specify scenarios in which control is wanted, we can let players share control on a vertex. Thus, buying strategies may involve regular expressions that specify conditions on the history of the computation, and the suggested prices depend on these conditions. For example, a user may be willing to pay for an edge that guarantees a certain service only after certain events have happened.

Pricing and deviations. In our setting, payments are made for all the sold edges. It is not hard to see that stability can be increased by charging players only for edges that actually participate in the outcome of the profile. On the other hand, the latter charging policy encourages players to bid for more edges. Also, in our setting, a player can deviate from a profile only if unilaterally changing her buying or selling strategies increases her utility. This deviation rule prevents players from initiating a trade, even if both the seller and buyer benefit from it. This motivates the definition of joined deviations, where, for example, two players can deviate together by offering and accepting an offer, respectively, as long as they both increase their utilities.

Game graphs. The fact our games are turned-based makes the ownership of control simple: Player i controls and may sell the vertices in V_i. It is possible, however, to trade control also in *concurrent* games. There, the movement of the token depends on actions taken by all the players in all the vertices. Two natural ways to trade control in a concurrent setting are transverse – when players buy the right to choose an action for the seller in certain vertices, or longitudinal – when each player has a set of variables she controls, and an action amounts to assigning values to these variables. Then, players may buy variables, namely the right to assign values to these variable throughout the computation. For example, in a system with users that direct robots in warehouse by assigning them a direction and speed, a user may sell the control on her robot in certain locations in the warehouse, or sell the ability to decide its speed throughout the computation. Finally, as in other game-graphs studied in formal methods, it is interesting to study extensions to richer settings, addressing incomplete information, infinite domains, stochastic behavior, and more.

References

1 M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 25th Int. Colloq. on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1989.

2 S.V. Albrecht and M.J. Wooldridge. Multi-agent systems research in the united kingdom. *AI Commun.*, 35(4):269–270, 2022.

3 S. Almagor, U. Boker, and O. Kupferman. Formalizing and reasoning about quality. *Journal of the ACM*, 63(3):24:1–24:56, 2016.

4 S. Almagor, D. Kuperberg, and O. Kupferman. Sensing as a complexity measure. *Int. J. Found. Comput. Sci.*, 30(6-7):831–873, 2019.

5 S. Almagor and O. Kupferman. High-quality synthesis against stochastic environments. In *Proc. 25th Annual Conf. of the European Association for Computer Science Logic*, volume 62 of *LIPIcs*, pages 28:1–28:17, 2016.

6 S. Almagor, O. Kupferman, and G. Perelli. Synthesis of controllable Nash equilibria in quantitative objective game. In *Proc. 27th Int. Joint Conf. on Artificial Intelligence*, pages 35–41, 2018.

7 M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *Proc. of 32nd Conf. on Artificial Intelligence*, pages 2669–2678. AAAI Press, 2018.

8 E. Anshelevich, A. Dasgupta, J. Kleinberg, E. Tardos, T. Wexler, and T. Roughgarden. The price of stability for network design with fair cost allocation. In *Proc. 45th IEEE Symp. on Foundations of Computer Science*, pages 295–304. IEEE Computer Society, 2004.

9 G. Avni, R. Bloem, K. Chatterjee, T. A. Henzinger, B. Könighofer, and S. Pranger. Run-time optimization for learned controllers through quantitative games. In *Proc. 31st Int. Conf. on Computer Aided Verification*, volume 11561 of *Lecture Notes in Computer Science*, pages 630–649. Springer, 2019.

10 G. Avni, P. Ghorpade, and S. Guha. A game of pawns, 2023. `arXiv:2305.04096`.

11 G. Avni and T.A. Henzinger. An updated survey of bidding games on graphs. In *47th Int. Symp. on Mathematical Foundations of Computer Science*, volume 241 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:6. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

12 G. Avni, T.A. Henzinger, and V. Chonev. Infinite-duration bidding games. *Journal of the ACM*, 66(4):31:1–31:29, 2019.

13 G. Avni, T.A. Henzinger, and D. Zikelic. Bidding mechanisms in graph games. *J. Comput. Syst. Sci.*, 119:133–144, 2021.

14 G. Avni, I. Jecker, and D. Zikelic. Infinite-duration all-pay bidding games. In *Symposium on Discrete Algorithms*, pages 617–636. SIAM, 2021.

15 G. Avni and O. Kupferman. Synthesis from component libraries with costs. In *Proc. 25th Int. Conf. on Concurrency Theory*, volume 8704 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2014.

16 BBC. Should billboard advertising be banned? `https://www.bbc.com/news/business-62806697`, 2022.

17 R. Bloem, K. Chatterjee, and B. Jobstmann. Graph games and reactive synthesis. In *Handbook of Model Checking.*, pages 921–962. Springer, 2018.

18 P. Bouyer-Decitre, O. Kupferman, N. Markey, B. Maubert, A. Murano, and G. Perelli. Reasoning about quality and fuzziness of strategic behaviours. In *Proc. 28th Int. Joint Conf. on Artificial Intelligence*, pages 1588–1594, 2019.

19 V. Bruyère. Synthesis of equilibria in infinite-duration games on graphs. *ACM SIGLOG News*, 8(2):4–29, 2021.

20 V. Bruyère. *A Game-Theoretic Approach for the Synthesis of Complex Systems*, pages 52–63. Springer International Publishing, 2022.

21 K. Chatterjee, T. A. Henzinger, and N. Piterman. Generalized parity games. In *Proc. 10th Int. Conf. on Foundations of Software Science and Computation Structures*, volume 4423 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2007.

22 K. Chatterjee, R. Majumdar, and T. A. Henzinger. Controller synthesis with budget constraints. In *Proc 11th International Workshop on Hybrid Systems: Computation and Control*, volume 4981 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2008.

23 S. Chaudhuri, S. Kannan, R. Majumdar, and M.J. Wooldridge. Game theory in AI, logic, and algorithms (dagstuhl seminar 17111). *Dagstuhl Reports*, 7(3):27–32, 2017.

24 R. Condurache, E. Filiot, R. Gentilini, and J.-F. Raskin. The complexity of rational synthesis. In *Proc. 43th Int. Colloq. on Automata, Languages, and Programming*, volume 55 of *LIPIcs*, pages 121:1–121:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

25 R. Condurache, Y. Oualhadj, and N. Troquard. The Complexity of Rational Synthesis for Concurrent Games. In *Proc. 29th Int. Conf. on Concurrency Theory*, volume 118 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:15, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

26 D. Fisman, O. Kupferman, and Y. Lustig. Rational synthesis. In *Proc. 16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2010.

27 O. Kupferman. Examining classical graph-theory problems from the viewpoint of formal-verification methods. In *Proc. 49th ACM Symp. on Theory of Computing*, page 6, 2017.

28 O. Kupferman, G. Perelli, and M.Y. Vardi. Synthesis with rational environments. *Annals of Mathematics and Artificial Intelligence*, 78(1):3–20, 2016.

29 O. Kupferman and N. Piterman. Lower bounds on witnesses for nonemptiness of universal co-Büchi automata. In *Proc. 12th Int. Conf. on Foundations of Software Science and Computation Structures*, volume 5504 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2009.

30 O. Kupferman and N. Shenwald. On the complexity of LTL rational synthesis. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *Lecture Notes in Computer Science*, pages 25–45, 2022.

31 Meta. Introduction to the advertising standards. `https://transparency.fb.com/policies/ad-standards/`, 2023.

32 J.F. Nash. Equilibrium points in n-person games. In *Proceedings of the National Academy of Sciences of the United States of America*, 1950.

33 N. Nisan, T. Roughgarden, E. Tardos, and V.V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

34 C. H. Papadimitriou. Algorithms, games, and the internet. In *Proc. 33rd ACM Symp. on Theory of Computing*, pages 749–753, 2001.

35 A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, 1989.

36 M. Ummels. The complexity of Nash equilibria in infinite multiplayer games. In *Proc. 11th Int. Conf. on Foundations of Software Science and Computation Structures*, pages 20–34, 2008.

37 J. von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton University Press, 1953.

Expressiveness Results for an Inductive Logic of Separated Relations

Radu Iosif
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000, France

Florian Zuleger
Institute of Logic and Computation, Technische Universität Wien, Austria

Abstract

In this paper we study a Separation Logic of Relations (SLR) and compare its expressiveness to (Monadic) Second Order Logic [(M)SO]. SLR is based on the well-known Symbolic Heap fragment of Separation Logic, whose formulæ are composed of points-to assertions, inductively defined predicates, with the separating conjunction as the only logical connective. SLR generalizes the Symbolic Heap fragment by supporting general relational atoms, instead of only points-to assertions. In this paper, we restrict ourselves to finite relational structures, and hence only consider Weak (M)SO, where quantification ranges over finite sets. Our main results are that SLR and MSO are incomparable on structures of unbounded treewidth, while SLR can be embedded in SO in general. Furthermore, MSO becomes a strict subset of SLR, when the treewidth of the models is bounded by a parameter and all vertices attached to some hyperedge belong to the interpretation of a fixed unary relation symbol. We also discuss the problem of identifying a fragment of SLR that is equivalent to MSO over models of bounded treewidth.

2012 ACM Subject Classification Theory of computation → Separation logic

Keywords and phrases Separation Logic, Model Theory, Monadic Second Order Logic, Treewidth

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.20

Related Version *Full Version*: `https://arxiv.org/abs/2307.02381` [42]

Funding *Radu Iosif*: The first author wishes to acknowledge the support of the French National Research Agency project Non-Aggregative Resource COmpositions (NARCO) under grant number ANR-21-CE48-0011.

1 Introduction

Relational structures are interpretations of relation symbols that define the standard semantics of first and second order logic [58]. They provide a unifying framework for reasoning about a multitude of graph types e.g., graphs with multiple edges, labeled graphs, colored graphs, hypergraphs, etc. Graphs are, in turn, important for many areas of computing, e.g., static analysis [45], databases and knowledge representation [1] and concurrency [27].

A well-established language for specifying graph properties is Monadic Second Order Logic (MSO), where quantification is over vertices only, or both vertices and edges, and sets thereof [25]. Other graph description logics use formal language theory (e.g., regular expressions, context-free grammars) to check for paths with certain patterns [37].

Another way of describing graphs is by an algebra of operations, such as vertex/hyperedge replacement, i.e., substitution of a vertex/hyperedge in a graph by another graph. Graph algebras come with robust notions of *recognizable sets* (i.e., unions of equivalence classes of a finite index congruence) and *inductive sets* (i.e., least solutions of recursive sets of equations, sometimes also called *equational* or *context-free* sets [25]). The relation between the expressivity of MSO-definable, recognizable and inductive sets is well-understood: all definable sets are recognizable, but there are recognizable sets that are not definable [22].

The equivalence between definability and recognizability has been established for those sets in which the *treewidth* (a positive integer that indicates how close the graph is to a tree) is bounded by a fixed constant [6]. Moreover, it is known that the set of graphs of treewidth bounded by a constant is inductive [25, Theorem 2.83].

From a system designer's point of view, logical specification is declarative (i.e., it describes required properties, such as acyclicity, hamiltonicity, etc.), whereas algebraic specification is operational (i.e., describes the way graphs are built from pieces), relying on low-level details (e.g., designated source vertices). Because of this, system provers (e.g., model checkers or deductive verifiers) tend to use logic both for requirement specification and internal representation of configuration sets. However, algebraic theories (e.g., automata theory) are used to obtain algorithms for discharging the generated logical verification conditions, e.g., satisfiability of formulæ or validity of entailments between formulæ.

Separation Logic (SL) [43, 56, 18] is a first order substructural logic with a *separating conjunction* $*$ that decomposes structures. For reasons related to its applications in the deductive verification of pointer-manipulating programs, the models of SL are finite graphs of fixed outdegree, described by partial functions, called *heaps*. The separating conjunction is interpreted in SL as the union of heaps with disjoint domains.

Since their early days, substructural logics have had (abstract) algebraic semantics [54], yet their relation with graph algebras has received scant attention. However, as we argue in this paper, the standard interpretation of the separating conjunction has the flavor of certain graph-algebraic operations, such as the disjoint union with fusion of designated nodes [23].

The benefits of SL over purely boolean graph logics (e.g., MSO) are two-fold:

I. The separating conjunction in combination with inductive definitions [2] provide concise descriptions of datastructures in the heap memory of a program. For instance, the rules (1) $\mathsf{ls}(x,y) \leftarrow x = y$ and (2) $\mathsf{ls}(x,y) \leftarrow \exists z \;.\; x \mapsto z * \mathsf{ls}(z,y)$ define finite singly-linked list segments, that are either (1) empty with equal endpoints, or (2) consist of a single cell x *separated from* the rest of the list segment $\mathsf{ls}(z,y)$. Most recursive datastructures (singly- and doubly-linked lists, trees, etc.) can be defined using only existentially quantified spatial conjunctions of atoms, that are (dis-)equalities and points-to atoms. This simple subset of SL is referred to as the *Symbolic Heap* fragment. The problems of model checking [13], satisfiability [12], robustness properties [44] and entailment [21, 47, 34, 35, 53] for this fragment have been studied extensively.

II. The separating conjunction is a powerful tool for reasoning about mutations of heaps. In fact, the built-in separating conjunction allows to describe actions *locally*, i.e., only with respect to the resources (e.g., memory cells, network nodes) involved, while framing out the part of the state that is irrelevant for that particular action. This principle of describing mutations, known as *local reasoning* [16], is at the heart of very powerful compositional proof techniques for pointer programs using SL [14].

The extension of SL from heaps to relational structures, called *Separation Logic of Relations* (SLR), has been first considered for relational databases and type systems of object-oriented languages, known as role logic [48]. Our motivation for studying the expressivity of SLR arose from several works:

(1) deductive verification of self-adapting distributed systems, where Hoare-style local reasoning is applied to write correctness proofs for systems with dynamically changing network architectures [4, 7, 9], and

(2) model-checking such systems for absence of deadlocks and critical section violations [10].

Another possible application of SLR is reasoning about programs with overlaid datastructures [31, 46], using variants of SL with a per-field composition of heaps, naturally expressed in SLR.

Table 1 A comparison of SLR, MSO and SO in terms of expressiveness, where ✓ means that the inclusion holds, × means it does not and ? denotes an open problem.

	SLR		MSO		SO	
SLR	✓	?	× (§4)	× (§4)	✓(§5)	✓(§7)
MSO	× (§4)	✓(§6)	✓	✓(§7)	✓	✓(§7)
SO	× (§4)	?	× (§7)	× (§7)	✓	✓(§7)

The SLR separating conjunction is understood as splitting the interpretation of each relation symbol from the signature into disjoint parts. For instance, the formula $\mathsf{R}(x_1, \ldots, x_n)$ describes a structure in which all relations are empty and R consists of a single tuple of values $x_1, \ldots, x_n$, whereas $\mathsf{R}(x_1, \ldots, x_n) * \mathsf{R}(y_1, \ldots, y_n)$ says that R consists of two distinct tuples, i.e., the values of x_i and y_i differ for at least one index $1 \leq i \leq n$. In contrast to the Courcelle-style composition of disjoint structures with fusion of nodes that interpret the common constants (i.e., function symbols of arity zero) [23], the SLR-style composition (i.e., the pointwise disjoint union of the interpretations of each relation symbol) is more fine-grained. For instance, if structures are used to encode graphs, SLR allows to specify (hyper-)edges that have no connected vertices, isolated vertices, or both. The same style of composition is found in other spatial logics for graphs, such as the GL logic of Cardelli, Gardner and Ghelli [18].

In particular, SLR is strictly more expressive than standard SL interpreted over heaps. For instance, the previous definition of a list segment can be written in a relational signature having at least a unary relation $\mathfrak{D}$ and a binary relation $\mathfrak{H}$, as (1) $\mathsf{rls}(x, y) \leftarrow x = y$ and (2) $\mathsf{rls}(x, y) \leftarrow \exists z \,.\, \mathfrak{D}(x) * \mathfrak{H}(x, z) * \mathsf{rls}(z, y)$. Note that the $\mathfrak{D}(x)$ atoms joined by separating conjunction ensure that all the nodes are pairwise different, except for the last one denoted by y. We will later generalize this use of $\mathfrak{D}$ for the definition of a Courcelle-style composition operator [23], where $\mathfrak{D}$ ensures that all but a bounded number of nodes are pairwise different. Further, SLR can describe graphs of unbounded degree, e.g., stars with a central vertex and outgoing binary edges E to frontier vertices e.g., (1) $\mathsf{star}(x) \leftarrow \mathfrak{N}(x) * \mathsf{node}(x)$ (2) $\mathsf{node}(x) \leftarrow x = x$ and (3) $\mathsf{node}(x) \leftarrow \exists y \,.\, \mathfrak{E}(x, y) * \mathfrak{N}(y) * \mathsf{node}(x)$. The definition of stars is not possible with SL interpreted over heaps, because of their bounded out-degree.

Our contributions. We compare the expressiveness of SLR with (monadic) second-order logic (M)SO. We are interested in finite relational structures, and hence only consider *weak* (M)SO, where relations are interpreted as finite sets.

For a logic $\mathcal{L} \in \{\mathsf{SLR}, \mathsf{MSO}, \mathsf{SO}\}$ using a finite set Σ of relation and constant symbols, we denote by $\llbracket \mathcal{L} \rrbracket$ the set of sets of models for all formulæ $\phi \in \mathcal{L}$. For a unary relation symbol $\mathfrak{D}$ not in Σ, considered fixed in the rest of the paper, we say that a graph is *guarded* if all elements from a tuple in the interpretation of a relation symbol belong to the interpretation of $\mathfrak{D}$. Then $\llbracket \mathcal{L} \rrbracket^{\mathfrak{D},k}$ is the set of sets of guarded models of treewidth at most k of a formula from $\mathcal{L}$, where the signature of $\mathcal{L}$ is extended with $\mathfrak{D}$, and $\llbracket \mathcal{L}_1 \rrbracket^{\mathfrak{D},k} \subseteq \llbracket \mathcal{L}_2 \rrbracket$ means that $\mathcal{L}_2$ is at least as expressive as $\mathcal{L}_1$, when only guarded models of treewidth at most k are considered. Note that $\llbracket \mathcal{L} \rrbracket^{\mathfrak{D},k} \subseteq \llbracket \mathcal{L} \rrbracket$ is not a trivial statement, in general, because it asserts the existence of a formula of $\mathcal{L}$ that defines the set of guarded structures of treewidth at most k.

Each cell of Table 1 shows $\llbracket \mathcal{L}_1 \rrbracket \subseteq \llbracket \mathcal{L}_2 \rrbracket$ (left) and $\llbracket \mathcal{L}_1 \rrbracket^{\mathfrak{D},k} \subseteq \llbracket \mathcal{L}_2 \rrbracket$ (right). Here ✓ means that the inclusion holds, × means it does not and ? denotes an open problem, with reference to the sections where (non-trivial) proofs are given. The most interesting cases are:

1. SLR and MSO are incomparable on unguarded structures of unbounded treewidth, i.e., there are formulæ in each of the logics that do not have an equivalent in the other,

2. SO is strictly more expressive than SLR, when considering unguarded structures of unbounded treewidth, and at least as expressive as SLR, when considering guarded structures of bounded treewidth,
3. SLR is strictly more expressive than MSO, when considering guarded structures of bounded treewidth; this shows the expressive power of SLR, emphasizing (once more) the model-theoretic importance of the treewidth parameter.

Note that, when considering SLR-definable sets of bounded treewidth, we systematically assume these structures to be guarded. We state as an open problem and conjecture that every infinite SLR-definable set of structures of bounded treewidth is necessarily guarded, in a hope that the guardedness condition can actually be lifted. So far, similar conditions have been used to, e.g., obtain decidability of entailments between SL symbolic heaps [41, 47] and of invariance for assertions written in a fragment of SLR for verifying distributed networks [9]. Moreover, the problem of checking if a given set of inductive definitions defines a guarded set of structures is decidable for these logics [44, 8].

A further natural question asks for a fragment of SLR with the same expressive power as MSO, over structures of bounded treewidth. This is also motivated by the need for a general fragment of SLR with a decidable entailment problem, that is instrumental in designing automated verification systems. Unfortunately, such a definition is challenging because the MSO-definability of the sets defined by SLR is an undecidable problem, whereas treewidth boundedness of such sets remains an open problem, conjectured to be decidable.

All proofs can be found in the full version of the paper [42].

Related work. Treewidth is a cornerstone of algorithmic tractability. For instance, many NP-complete graph problems such as Hamiltonicity and 3-Coloring become PTIME, when restricted to inputs whose treewidth is bounded by a constant, see, e.g., [38, Chapter 11]. Moreover, bounding the treewidth by a constant sets the frontier between the decidability and undecidability of monadic second order (MSO) logical theories. A result of Courcelle [22] proves that MSO is decidable over bounded treewidth structures, by reduction to the emptiness problem of tree automata. A dual result of Seese [57] proves that each class of structures with a decidable MSO theory necessarily has bounded treewidth.

Comparing the expressiveness of SL [56] with classical logics received a fair amount of attention. A first proof of undecidability of the satisfiability problem for SL, with first order quantification, negation and separating implication, but without inductive definitions [17], is based on a reduction to Trakhtenbrot's undecidability result for first order logic on finite models [32]. This proof uses heaps of outdegree two to encode arbitrary binary relations as $\mathsf{R}(x, y) \stackrel{\mathsf{def}}{=} \exists z \;.\; z \mapsto (x, y) * \mathsf{true}$. A more refined proof for heaps of outdegree one was given in [11], where it was shown that SO has the same expressivity as SL, when negation and separating implication is allowed, which is not the case for our fragment of SLR.

A related line of work, pioneered by Lozes [50], is the translation of quantifier-free SL formulæ into boolean combinations of *core formulæ*, belonging to a small set of very simple patterns. This enables a straightforward translation of the quantifier-free fragment of SL into first order logic, over unrestricted signatures with both relation and function symbols, subsequently extended to two quantified variables [28] and restricted quantifier prefixes [33]. Moreover, a translation of quantifier-free SL into first order logic, based on the small model property of the former, has been described in [15]. These are fragments of SL without inductive definitions, but with arbitrary combinations of boolean (conjunction, negation) and spatial (separating conjunction, magic wand) connectives. A non-trivial attempt of generalizing the technique of core formulæ to *reachability* and *list segment* predicates is given in [29]. Moreover, an in-depth comparison between the expressiveness of various models of separation, i.e., spatial, as in SL, and contextual (subtree-like), as in Ambient Logic [19], can be found in [52]. The restriction of SLR on trees is, however, out of the scope of this paper.

An early combination of spatial connective for graph decomposition with (least fixpoint) recursion is Graph Logic (GL) [18], whose expressiveness is compared to that of MSO_2, i.e., MSO interpreted over graphs, with quantification over both vertices and edges [26]. For reasons related to its applications, GL quantifies over the vertices and edge labels of a graph, unlike MSO_2 that quantifies over vertices, edges and sets thereof. Another fairly subtle difference is that GL can describe graphs with multiple edges that involve the same vertices and same label, whereas the models of MSO_2 are simple graphs. Without recursion, GL can be translated into MSO_2 and it has been shown that MSO_2 is strictly more expressive than GL without edge label quantification [5]. Little is known for GL with recursion, besides that it can express PSPACE-complete model checking problems [26], whereas model checking is PSPACE-complete for MSO [59].

The separating conjunction used in SLR has been first introduced in role logic [48], a logic designed to reason about properties of record fields in object-oriented programs. This logic uses separating conjunction in combination with boolean connectives and first order quantifier (ranging over vertices) and has no recursive constructs (least fixpoints or inductive definitions). A bothways translation between role logic and SO has been described in [49]. These translations rely on boolean connectives and first order quantifiers, instead of least fixpoint recursion, which is the case in our work.

To complete the picture, a substructural logic with separating conjunction and implication, based on a layered decomposition of graphs has been developped in [20]. However, the relation between this logic and (M)SO remains unexplored, to the best of our knowledge.

2 Definitions

For a set A, we denote by $\mathrm{pow}(A)$ its powerset, $A^1 \stackrel{\text{def}}{=} A$, $A^{i+1} \stackrel{\text{def}}{=} A^i \times A$, for all $i \geq 1$, where $\times$ is the Cartesian product, and $A^+ \stackrel{\text{def}}{=} \bigcup_{i\geq 1} A^i$. The cardinality of a finite set A is denoted by $\|A\|$. Given integers i and j, we write $[i, j]$ for the set $\{i, i+1, \ldots, j\}$, empty if $i > j$. For a partial function $f : A \to B$, we denote by $\mathrm{dom}(f)$ its domain and by $f\downharpoonright_S$ its restriction to $S \subseteq \mathrm{dom}(f)$. f is *locally co-finite* iff the set $\{a \in A \mid f(a) = b\}$ is finite, for all $b \in B$. f is *effectively computable* iff there exists a Turing machine $\mathcal{M}$, such that, for any $a \in \mathrm{dom}(f)$, $\mathcal{M}$ outputs $f(a)$ in finitely many steps and diverges for $a \not\in \mathrm{dom}(f)$.

Signatures and Structures. Let $\Sigma = \{\mathsf{R}_1, \ldots, \mathsf{R}_N, \mathsf{c}_1, \ldots, \mathsf{c}_M\}$ be a finite *signature*, where R_i are relation symbols of arity $\#\mathsf{R}_i \geq 1$ and c_j are constant symbols, i.e., function symbols of arity zero. Additionally, we assume the existence of a unary relation symbol $\mathfrak{D}$, not in Σ. Unless stated otherwise, we consider Σ and $\mathfrak{D}$ to be fixed in the following.

A *structure* is a pair (U, σ), where U is an *infinite* set, called *universe*, and $\sigma : \Sigma \to U \cup \mathrm{pow}(U^+)$ is an *interpretation* that maps each relation symbol R to a relation $\sigma(\mathsf{R}) \subseteq U^{\#\mathsf{R}}$ and each constant c to an element $\sigma(\mathsf{c}) \in U$. Two structures are *isomorphic* iff they differ only by a renaming of their elements (a formal definition is given in, e.g., [32, §A3]). We write $\mathrm{Rel}(\sigma)$ for the set of elements that belong to $\sigma(\mathsf{R})$, for some relation symbol $\mathsf{R} \in \Sigma$ and $\mathrm{Supp}(\sigma) \stackrel{\text{def}}{=} \mathrm{Rel}(\sigma) \cup \{\sigma(\mathsf{c}_1), \ldots, \sigma(\mathsf{c}_M)\}$ for the *support* of the structure, that includes the interpretation of constants. We denote by $\mathit{Str}(\Sigma)$ (resp. $\mathit{Str}(\Sigma, \mathfrak{D})$) the set of structures over the signature Σ (resp. $\Sigma \cup \{\mathfrak{D}\}$).

A structure is *guarded* iff all nodes that occur in some tuple from the denotation of a relation symbol sit also inside the denotation of the unary relation $\mathfrak{D}$:

▶ **Definition 1.** *A structure* $(U, \sigma) \in \mathit{Str}(\Sigma, \mathfrak{D})$ *is* guarded *iff* $\mathrm{Rel}(\sigma) = \sigma(\mathfrak{D})$.

Two interpretations σ_1 and σ_2 are *compatible* iff $\sigma_1(\mathsf{c}) = \sigma_2(\mathsf{c})$, for all constant symbols $\mathsf{c} \in \Sigma$. Two structures (U_1, σ_1) and (U_2, σ_2) are *locally disjoint* iff $\sigma_1(\mathsf{R}) \cap \sigma_2(\mathsf{R}) = \emptyset$, for all relation symbols $\mathsf{R} \in \Sigma$. The (spatial) *composition* of structures is defined below:

▶ **Definition 2.** *The* composition *of two compatible and locally disjoint structures* (U_1, σ_1) *and* (U_2, σ_2) *is* $(U_1, \sigma_1) \bullet (U_2, \sigma_2) \stackrel{\text{def}}{=} (U_1 \cup U_2, \sigma_1 \uplus \sigma_2)$*, where* $(\sigma_1 \uplus \sigma_2)(\mathsf{R}_i) \stackrel{\text{def}}{=} \sigma_1(\mathsf{R}_i) \cup \sigma_2(\mathsf{R}_i)$ *and* $(\sigma_1 \uplus \sigma_2)(\mathsf{c}_j) \stackrel{\text{def}}{=} \sigma_1(\mathsf{c}_j) = \sigma_2(\mathsf{c}_j)$*, for all* $i \in [1, N]$ *and* $j \in [1, M]$*. The composition is undefined for structures that are not compatible or not locally disjoint.*

Graphs and Treewidth. A graph is a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, such that $\mathcal{V}$ is a set of *vertices* and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of *edges*. All graphs considered in this paper are finite and directed, i.e., $\mathcal{E}$ is not necessarily a symmetric relation. Graphs are naturally encoded as structures:

▶ **Definition 3.** *A graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *is encoded by the structure* $(U_{\mathcal{G}}, \sigma_{\mathcal{G}})$ *over the signature* $\Gamma \stackrel{\text{def}}{=} \{\mathfrak{V}, \mathfrak{E}\}$*, where* $\#\mathfrak{V} = 1$ *and* $\#\mathfrak{E} = 2$*, such that* $U_{\mathcal{G}} = \mathcal{V}$*,* $\sigma_{\mathcal{G}}(\mathfrak{V}) = \mathcal{V}$ *and* $\sigma_{\mathcal{G}}(\mathfrak{E}) = \mathcal{E}$.

A *path* in $\mathcal{G}$ is a sequence of pairwise distinct vertices $v_1, \ldots, v_n$, such that $(v_i, v_{i+1}) \in \mathcal{E}$, for all $i \in [1, n-1]$. We say that $v_1, \ldots, v_n$ is an *undirected path* if $\{(v_i, v_{i+1}), (v_{i+1}, v_i)\} \cap \mathcal{E} \neq \emptyset$ instead, for all $i \in [1, n-1]$. A set of vertices $V \subseteq \mathcal{V}$ is *connected in* $\mathcal{G}$ iff there is an undirected path in $\mathcal{G}$ between any two vertices in V. A graph $\mathcal{G}$ is *connected* iff $\mathcal{V}$ is connected in $\mathcal{G}$. A *clique* is a graph such that each two distinct nodes are the endpoints of an edge, the direction of which is not important. We denote by $\mathcal{K}_n$ the set of cliques with n vertices.

Given a set Λ of labels, a Λ-*labeled tree* is a tuple $\mathcal{T} = (\mathcal{N}, \mathcal{F}, r, \lambda)$, where $(\mathcal{N}, \mathcal{F})$ is a graph, $r \in \mathcal{N}$ is a designated vertex called the *root*, such that there exists a unique path in $(\mathcal{N}, \mathcal{F})$ from r to any other vertex $v \in \mathcal{N} \setminus \{r\}$ and r has no incoming edges $(p, r) \in \mathcal{F}$. The mapping $\lambda : \mathcal{N} \rightarrow \Lambda$ associates each vertex of the tree a *label* from Λ.

▶ **Definition 4.** *A* tree decomposition *of a structure* (U, σ) *over the signature* Σ *is a* $\mathrm{pow}(U)$-*labeled tree* $\mathcal{T} = (\mathcal{N}, \mathcal{F}, r, \lambda)$*, such that the following hold:*

1. *for each relation symbol* $\mathsf{R} \in \Sigma$ *and each tuple* $\langle u_1, \ldots, u_{\#\mathsf{R}} \rangle \in \sigma(\mathsf{R})$ *there exists* $n \in \mathcal{N}$*, such that* $\{u_1, \ldots, u_{\#\mathsf{R}}\} \subseteq \lambda(n)$*, and*
2. *for each* $u \in \mathrm{Supp}(\sigma)$*, the set* $\{n \in \mathcal{N} \mid u \in \lambda(n)\}$ *is nonempty and connected in* $(\mathcal{N}, \mathcal{F})$.

The width *of the tree decomposition is* $\mathrm{tw}(\mathcal{T}) \stackrel{\text{def}}{=} \max_{n \in \mathcal{N}} \|\lambda(n)\| - 1$*. The* treewidth *of the structure* (U, σ) *is* $\mathrm{tw}(U, \sigma) \stackrel{\text{def}}{=} \min\{\mathrm{tw}(\mathcal{T}) \mid \mathcal{T}$ *is a tree decomposition of* $\sigma\}$.

A set of structures is *treewidth-bounded* iff the set of corresponding treewidths is finite and *treewidth-unbounded* otherwise. A set is *strictly treewidth-unbounded* iff it is treewidth-unbounded and any of its infinite subsets is treewidth-unbounded. The following result can be found in [30, Theorem 12.3.9] and is restated here for self-containment:

▶ **Proposition 5.** *The set of cliques* $\{\mathcal{K}_n \mid n \in \mathbb{N}\}$ *is strictly treewidth-unbounded.*

3 Logics

We introduce two logics over a relational signature $\Sigma = \{\mathsf{R}_1, \ldots, \mathsf{R}_N, \mathsf{c}_1, \ldots, \mathsf{c}_M\}$. First, the *Separation Logic of Relations* (SLR) uses a set of *first order variables* $\mathbb{V}_1 = \{x, \ldots\}$ and a set of *predicates* $\mathbb{A} = \{\mathsf{A}, \ldots\}$ (also called *recursion variables* in the literature, e.g., [18]) of arities $\#\mathsf{A} \geq 0$. We use the symbols $\xi, \chi \in \mathbb{V}_1 \cup \{\mathsf{c}_1, \ldots, \mathsf{c}_M\}$ to denote *terms*, i.e., either first order variables or constants. The formulæ of SLR are defined by the following syntax:

$$\phi := \mathsf{emp} \mid \xi = \chi \mid \xi \neq \chi \mid \mathsf{R}(\xi_1, \ldots, \xi_{\#\mathsf{R}}) \mid \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}}) \mid \phi * \phi \mid \exists x \,.\, \phi$$

The formulæ $\xi = \chi$ and $\xi \neq \chi$ are called *equalities* and *disequalities*, $\mathsf{R}(\xi_1, \ldots, \xi_{\#\mathsf{R}})$ and $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$ are called *relation* and *predicate atoms*, respectively. A formula with no occurrences of predicate atoms (resp. existential quantifiers) is called *predicate-free* (resp. *quantifier-free*). A variable is *free* if it does not occur within the scope of an existential quantifier and *bound* otherwise. We denote by $\mathrm{fv}(\phi)$ be the set of free variables of ϕ. A *sentence* is a formula with no free variables. A *substitution* $\phi[x_1/\xi_1 \ldots x_n/\xi_n]$ replaces simultaneously every occurrence of the free variable x_i by the term ξ_i in ϕ, for all $i \in [1, n]$. As a convention, the bound variables in ϕ are renamed to avoid clashes with $\xi_1, \ldots, \xi_n$.

The predicates from $\mathbb{A}$ are interpreted as sets of structures, defined inductively:

▶ **Definition 6.** *A* set of inductive definitions (SID) Δ *is a* finite *set of* rules *of the form* $\mathsf{A}(x_1, \ldots, x_{\#\mathsf{A}}) \leftarrow \phi$*, where* $x_1, \ldots, x_{\#\mathsf{A}}$ *are pairwise distinct variables, called* parameters*, such that* $\mathrm{fv}(\phi) \subseteq \{x_1, \ldots, x_{\#\mathsf{A}}\}$*. A rule* $\mathsf{A}(x_1, \ldots, x_{\#\mathsf{A}}) \leftarrow \phi$ *is said to* define A.

The semantics of SLR formulæ is given by the satisfaction relation $(U, \sigma) \models^{\nu}_{\Delta} \phi$ between structures and formulæ. This relation is parameterized by a *store* $\nu : \mathbb{V}_1 \to U$ mapping the free variables of a formula into elements of the universe and an SID Δ. We write $\nu[x \leftarrow u]$ for the store that maps x into u and agrees with ν on all variables other than x. For a term ξ, we denote by $(\sigma, \nu)(\xi)$ the value $\sigma(\xi)$ if ξ is a constant, or $\nu(\xi)$ if ξ is a first-order variable. The satisfaction relation is the least relation that satisfies the following conditions:

$$\begin{array}{lcl}
(U,\sigma) \models^{\nu}_{\Delta} \mathsf{emp} & \Leftrightarrow & \sigma(\mathsf{R}) = \emptyset, \text{ for all } \mathsf{R} \in \Sigma \\
(U,\sigma) \models^{\nu}_{\Delta} \xi \sim \chi & \Leftrightarrow & (U,\sigma) \models^{\nu}_{\Delta} \mathsf{emp} \text{ and } (\sigma,\nu)(\xi) \sim (\sigma,\nu)(\chi), \text{ where } \sim \in \{=, \neq\} \\
(U,\sigma) \models^{\nu}_{\Delta} \mathsf{R}(\xi_1, \ldots, \xi_{\#\mathsf{R}}) & \Leftrightarrow & \sigma(\mathsf{R}) = \{\langle (\sigma,\nu)(\xi_1), \ldots, (\sigma,\nu)(\xi_{\#\mathsf{R}}) \rangle\} \\
 & & \qquad \text{and } \sigma(\mathsf{R}') = \emptyset, \text{for } \mathsf{R}' \in \Sigma \setminus \{\mathsf{R}\} \\
(U,\sigma) \models^{\nu}_{\Delta} \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}}) & \Leftrightarrow & (U,\sigma) \models^{\nu}_{\Delta} \phi[x_1/\xi_1, \ldots, x_{\#\mathsf{A}}/\xi_{\#\mathsf{A}}], \\
 & & \qquad \text{for some } \mathsf{A}(x_1, \ldots, x_{\#\mathsf{A}}) \leftarrow \phi \in \Delta \\
(U,\sigma) \models^{\nu}_{\Delta} \phi_1 * \phi_2 & \Leftrightarrow & \text{there exist structures } (U_1,\sigma_1) \text{ and } (U_2,\sigma_2), \text{ such that} \\
 & & (U,\sigma) = (U_1,\sigma_1) \bullet (U_2,\sigma_2) \text{ and } (U,\sigma_i) \models^{\nu}_{\Delta} \phi_i, \text{ for } i = 1,2 \\
(U,\sigma) \models^{\nu}_{\Delta} \exists x \,.\, \phi & \Leftrightarrow & (U,\sigma) \models^{\nu[x \leftarrow u]}_{\Delta} \phi, \text{ for some } u \in U
\end{array}$$

Note that every structure (U, σ), such that $(U, \sigma) \models^{\nu}_{\Delta} \phi$, interprets each relation symbol as a finite set of tuples, defined by a finite least fixpoint iteration over the rules in Δ. In particular, the assumption that each universe is infinite excludes the cases in which a SLR formula becomes unsatisfiable because the universe does not have enough elements to be assigned to the existentially quantified variables during the unfolding of the rules.

If ϕ is a sentence, the satisfaction relation does not depend on the store, in which case we write $(U, \sigma) \models_{\Delta} \phi$ and say that (U, σ) is a Δ-model of ϕ. We denote by $[\![\phi]\!]_{\Delta}$ the set of Δ-models of ϕ. We call $[\![\phi]\!]_{\Delta}$ an SLR-*definable* set. By $[\![\phi]\!]^{\mathfrak{D},k}_{\Delta}$ we denote the set of guarded structures (Def. 1) of treewidth at most k from $[\![\phi]\!]_{\Delta}$. We write $[\![\mathsf{SLR}]\!] \stackrel{\mathsf{def}}{=} \{[\![\phi]\!]_{\Delta} \mid \phi \text{ is a } \mathsf{SLR} \text{ formula}, \Delta \text{ is a SID}\}$ and $[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \stackrel{\mathsf{def}}{=} \{[\![\phi]\!]^{\mathfrak{D},k}_{\Delta} \mid \phi \text{ is a } \mathsf{SLR} \text{ formula}, \Delta \text{ is a SID}\}$. Below we show that SLR-definable sets are unions of isomorphic equivalence classes:

▶ **Proposition 7.** *Given isomorphic structures (U, σ) and (U', σ'), for any sentence ϕ of SLR and any SID Δ, we have $(U, \sigma) \models_\Delta \phi \Leftrightarrow (U', \sigma') \models_\Delta \phi$.*

The other logic is the *Weak Second Order Logic* (SO) defined using a set of *second order variables* $\mathbb{V}_2 = \{X, \ldots\}$, in addition to first order variables $\mathbb{V}_1$. We denote by $\#X$ the arity of a second order variable X. Terms and atoms are the same as in SLR. The formulæ of SO have the following syntax:

$$\psi := \xi = \chi \mid \mathsf{R}(\xi_1, \ldots, \xi_{\#\mathsf{R}}) \mid X(\xi_1, \ldots, \xi_{\#X}) \mid \neg\psi \mid \psi \wedge \psi \mid \exists x \,.\, \psi \mid \exists X \,.\, \psi$$

We write $\xi \neq \chi \stackrel{\text{def}}{=} \neg\xi = \chi$, $\psi_1 \vee \psi_2 \stackrel{\text{def}}{=} \neg(\neg\psi_1 \wedge \neg\psi_2)$, $\psi_1 \rightarrow \psi_2 \stackrel{\text{def}}{=} \neg\psi_1 \vee \psi_2$, $\forall x \,.\, \psi \stackrel{\text{def}}{=} \neg\exists x \,.\, \neg\psi$ and $\forall X \,.\, \psi \stackrel{\text{def}}{=} \neg\exists X \,.\, \neg\psi$. The Weak Monadic Second Order Logic (MSO) is the fragment of SO restricted to second-order variables of arity one. The Weak Existential Second Order Logic (ESO) is the fragment of SO consisting of formulæ of the form $\exists X_1 \ldots \exists X_n \,.\, \phi$, where ϕ has only first order quantifiers.

The semantics of SO is given by a relation $(U, \sigma) \Vdash^\nu \psi$, where the store $\nu : \mathbb{V}_1 \cup \mathbb{V}_2 \rightarrow U \cup \mathrm{pow}(U^+)$ maps each first-order variable $x \in \mathbb{V}_1$ to an element of the universe $\nu(x) \in U$ and each second-order variable $X \in \mathbb{V}_2$ to a finite relation $\nu(X) \subseteq U^{\#X}$. The satisfaction relation of SO is defined inductively on the structure of formulæ:

$$\begin{array}{lcl}
(U, \sigma) \Vdash^\nu \xi = \chi & \Leftrightarrow & (\sigma, \nu)(\xi) = (\sigma, \nu)(\chi) \\
(U, \sigma) \Vdash^\nu \mathsf{R}(\xi_1, \ldots, \xi_{\#\mathsf{R}}) & \Leftrightarrow & \langle (\sigma, \nu)(\xi_1), \ldots, (\sigma, \nu)(\xi_{\#\mathsf{R}}) \rangle \in \sigma(\mathsf{R}) \\
(U, \sigma) \Vdash^\nu X(\xi_1, \ldots, \xi_{\#X}) & \Leftrightarrow & \langle (\sigma, \nu)(\xi_1), \ldots, (\sigma, \nu)(\xi_{\#X}) \rangle \in \nu(X) \\
(U, \sigma) \Vdash^\nu \exists X \,.\, \psi & \Leftrightarrow & (U, \sigma) \Vdash^{\nu[X \leftarrow V]} \psi, \text{ for some finite set } V \subseteq U^{\#X}
\end{array}$$

The semantics of negation, conjunction and first-order quantification are standard and omitted for brevity. Note the difference between equalities and relation atoms in SLR and SO: in the former, equalities (relation atoms) hold in an empty (singleton) structure, whereas no such upper bounds on the cardinality of the model of an atom occur in SO.

However, SO can express upper bounds on the cardinality of the universe. Such formulæ are unsatisfiable under the assumption that the universe of each structure is infinite. We chose to keep the comparison between SLR and SO simple and not consider the general case of a finite universe, for the time being. A detailed study of SL interpreted over finite universe heaps, with arbitrary nesting of boolean and separating connectives but without inductive definitions is given in [33]. We plan to give a similar comparison in an extended version.

If ϕ is a sentence, we write $(U, \sigma) \Vdash \phi$ instead of $(U, \sigma) \Vdash^\nu \phi$ and define $[\![\phi]\!] \stackrel{\text{def}}{=} \{(U, \sigma) \mid (U, \sigma) \Vdash \phi\}$ and $[\![\phi]\!]^{\mathfrak{D},k}$ for the restriction of $[\![\phi]\!]$ to guarded structures of treewidth at most k. We call $[\![\phi]\!]$ an (M)SO-*definable* set. We write $[\![\text{(M)SO}]\!] \stackrel{\text{def}}{=} \{[\![\phi]\!] \mid \phi \text{ is a (M)SO formula}\}$ and $[\![\text{(M)SO}]\!]^{\mathfrak{D},k} \stackrel{\text{def}}{=} \{[\![\phi]\!]^{\mathfrak{D},k} \mid \phi \text{ is a (M)SO formula}\}$.

The aim of this paper is comparing the expressive powers of SLR, MSO and SO, with respect to the properties that can be defined in these logics. We are concerned with the problems $[\![\mathcal{L}_1]\!] \subseteq [\![\mathcal{L}_2]\!]$ and $[\![\mathcal{L}_1]\!]^{\mathfrak{D},k} \subseteq [\![\mathcal{L}_2]\!]$, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are any of the logics SLR, MSO and SO, respectively. In particular, for $[\![\mathcal{L}_1]\!]^{\mathfrak{D},k} \subseteq [\![\mathcal{L}_2]\!]$, we implicitly assume that $\mathcal{L}_1$ and $\mathcal{L}_2$ are sets of formulæ over the relational signature $\Sigma \cup \{\mathfrak{D}\}$. Table 1 summarizes our results, with references to the sections in the paper where the (non-trivial) proofs can be found, and the remaining open problems.

4 $[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \not\subseteq [\![\mathsf{MSO}]\!] \not\subseteq [\![\mathsf{SLR}]\!]$

The argument that shows $[\![\text{SLR}]\!]^{\mathfrak{D},k} \not\subseteq [\![\text{MSO}]\!]$ is that MSO cannot express the fact that the cardinality of a set is even [22, Proposition 6.2]. The SLR rules below state that the cardinality of $\mathfrak{R}$ is even, for a predicate A of arity zero:

$$\mathsf{A}() \leftarrow \exists x \exists y \;.\; \mathfrak{R}(x) * \mathfrak{R}(y) * A() \qquad \mathsf{A}() \leftarrow \mathsf{emp}$$

Note that every model of $\mathsf{A}()$ interprets $\mathfrak{R}$ as a set with an even number of disconnected elements and every other relation symbol by an empty set. The treewidth of such models is one, thus $[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \not\subseteq [\![\mathsf{MSO}]\!]$ for any $k \geq 1$, and we obtain $[\![\mathsf{SLR}]\!] \not\subseteq [\![\mathsf{MSO}]\!]$, in general.

The argument for $[\![\mathsf{MSO}]\!] \not\subseteq [\![\mathsf{SLR}]\!]$ is that the set of cliques is MSO-definable (actually, even first order definable) but not SLR-definable. First, the set $\{\mathcal{K}_n \mid n \in \mathbb{N}\}$ is defined by the following first order formula in the signature of graph encodings (Def. 3):

$$\forall x \forall y \;.\; \mathfrak{V}(x) \wedge \mathfrak{V}(y) \wedge x \neq y \rightarrow \mathfrak{E}(x, y) \vee \mathfrak{E}(y, x)$$

Since this set is strictly treewidth-unbounded (Prop. 5), it is sufficient to prove that SLR cannot define strictly treewidth-unbounded sets. More precisely, for each SLR sentence ϕ and SID Δ, we prove the existence of an integer $W \geq 1$, depending on ϕ and Δ alone, such that

(i) for each structure $(U, \sigma) \in [\![\phi]\!]_\Delta$ there exists a structure $(U, \overline{\sigma}) \in [\![\phi]\!]_\Delta$, of treewidth at most W, and

(ii) the function that maps (U, σ) into $(U, \overline{\sigma})$ is locally co-finite (Lemma 10).

Then each infinite SLR-definable set has an infinite treewidth-bounded subset, i.e., it is not strictly treewidth-unbounded (Prop. 12).

A first ingredient of the proof is that each SID can be transformed into an equivalent SID without equality constraints between variables:

▶ **Definition 8.** *A rule* $\mathsf{A}(x_1, \ldots, x_{\#\mathsf{A}}) \leftarrow \exists y_1 \ldots \exists y_n \;.\; \psi$*, where* ψ *is a quantifier-free formula, is* normalized *iff no equality atom* $x = y$ *occurs in* ψ*, for distinct variables* $x, y \in \{x_1, \ldots, x_{\#\mathsf{A}}\} \cup \{y_1, \ldots, y_n\}$*. An SID is* normalized *iff it contains only normalized rules.*

▶ **Lemma 9.** *Given an SID* Δ*, one can build a normalized SID* Δ' *such that, for each structure* σ *and each predicate atom* $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$*, we have* $(U, \sigma) \models_\Delta \exists \xi_{i_1} \ldots \exists \xi_{i_n} \;.\; \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}}) \Leftrightarrow (U, \sigma) \models_{\Delta'} \exists \xi_{i_1} \ldots \exists \xi_{i_n} \;.\; \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$*, where* $\{\xi_{i_1}, \ldots, \xi_{i_n}\} = \{\xi_1, \ldots, \xi_{\#\mathsf{A}}\} \cap \mathbb{V}_1$*.*

A consequence is that, in the absence of equality constraints, each existentially quantified variable instantiated by the inductive definition of the satisfaction relation can be assigned a distinct element of the universe. For instance, considering the rules $\mathsf{fold\_ls}(x_1) \leftarrow \mathsf{emp}$ and $\mathsf{fold\_ls}(x_1) \leftarrow \exists y \;.\; \mathfrak{H}(x_1, y) * \mathsf{fold\_ls}(y)$, the $\mathsf{fold\_ls}(x)$ formula defines an infinite set of graphs whose edges are given by the interpretation of a relation symbol $\mathfrak{H}$, such that there exists an Eulerian path visiting all edges exactly once, and all vertices possibly more than once. Since there are no equality constraints, each model of $\mathsf{fold\_ls}(x)$ can be expanded into an acyclic list that never visits the same vertex twice, except at the endpoints. This graph has treewidth two, if the endpoints coincide, and one otherwise.

Formally, we write $(U, \sigma) \overset{\bullet}{\models}{}^\nu_\Delta \phi$ iff the satisfaction relation $(U, \sigma) \models^\nu_\Delta \phi$ can be established by considering finite injective stores. The definition of $\overset{\bullet}{\models}{}^\nu_\Delta$ is the same as the one of $\models^\nu_\Delta$ (§3), except for the cases below:

$$\begin{aligned} (U, \sigma) \overset{\bullet}{\models}{}^\nu_\Delta \phi_1 * \phi_2 \;&\Leftrightarrow \text{there exist structures } (U_1, \sigma_1) \bullet (U_2, \sigma_2) = (U, \sigma) \text{, such that} \\ &\qquad U_1 \cap U_2 = \nu(\mathrm{fv}(\phi_1) \cap \mathrm{fv}(\phi_2)) \text{ and } (U_i, \sigma_i) \overset{\bullet}{\models}{}^{\nu\downharpoonright_{\mathrm{fv}(\phi_i)}}_\Delta \phi_i \text{, for } i = 1, 2 \\ (U, \sigma) \overset{\bullet}{\models}{}^\nu_\Delta \exists x \;.\; \phi \;&\Leftrightarrow (U, \sigma) \overset{\bullet}{\models}{}^{\nu[x \leftarrow u]}_\Delta \phi \text{, for some } u \in U \setminus \nu(\mathrm{fv}(\phi)) \end{aligned}$$

For instance, we have $(U, \sigma) \overset{\bullet}{\models}{}^\nu_\Delta \mathsf{fold\_ls}(x)$ only if $\sigma(\mathfrak{H})$ is a list of pairwise distinct elements.

▶ **Lemma 10.** *Given a normalized SID Δ, a predicate atom $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$, for each structure (U, σ) and a store ν, such that $(U, \sigma) \models^{\nu}_{\Delta} \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$, there exists a structure $(U, \overline{\sigma})$, such that $(U, \overline{\sigma}) \overset{\bullet}{\models}{}^{\nu}_{\Delta} \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$. Moreover, the function with domain $[\![\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})]\!]_{\Delta}$ that maps (U, σ) into the set of structures isomorphic with $(U, \overline{\sigma})$ is locally co-finite.*

We show that the models defined on injective stores have bounded treewidth:

▶ **Lemma 11.** *Given a normalized SID Δ and a predicate atom $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$, we have $\mathrm{tw}(\sigma) \leq W$, for each structure (U, σ) and store ν, such that $(U, \sigma) \overset{\bullet}{\models}{}^{\nu}_{\Delta} \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$, where $W \geq 1$ is a constant depending only on Δ.*

Note that proving Lemmas 10 and 11 for predicate atoms loses no generality, because for each formula ϕ, such that $\mathrm{fv}(\phi) = \{x_1, \ldots, x_n\}$, we can consider a predicate symbol A_ϕ of arity n and extend the SID by the rule $\mathsf{A}_\phi(x_1, \ldots, x_n) \leftarrow \phi$. The proof of $[\![\mathsf{MSO}]\!] \not\subseteq [\![\mathsf{SLR}]\!]$ relies on the following:

▶ **Proposition 12.** *Given a sentence ϕ and an SID Δ, $[\![\phi]\!]_{\Delta}$ is either finite or it has an infinite subset of bounded treewidth.*

5 $[\![\mathsf{SLR}]\!] \subseteq [\![\mathsf{SO}]\!]$

Since SLR and MSO are incomparable, it is natural to ask for a logic that subsumes both of them. In this section, we prove that SO is such a logic. Since MSO is a syntactic subset of SO, we have $[\![\mathsf{MSO}]\!] \subseteq [\![\mathsf{SO}]\!]$ trivially. We show that $[\![\mathsf{SLR}]\!] \subseteq [\![\mathsf{SO}]\!]$ using the fact that each model of a predicate atom in SLR is built according to a *finite unfolding tree* indicating the partial order in which the rules of the SID are used in the inductive definition of the satisfaction relation; in other words, unfolding trees are for SIDs what derivation trees are for context-free grammars. More precisely, any model of a SLR sentence can be decomposed into pairwise disjoint substructures, each being the model of the quantifier- and predicate-free subformula of a rule in the SID, such that there is a one-to-one mapping between the nodes of the tree and the substructures from the decomposition of the model. We use second order variables, interpreted as finite relations, to define the unfolding tree and the mapping between the nodes of the unfolding tree and the tuples in the interpretation of the relation symbols from the model. These second order variables are existentially quantified and the resulting SO formula describes the model, without the unfolding tree that witnesses its construction according to the rules of the SID.

Let $\Delta \overset{\mathsf{def}}{=} \{\mathsf{r}_1, \ldots, \mathsf{r}_R\}$ be a given SID. Without loss of generality, for each relation symbol $\mathsf{R} \in \Sigma$, we assume that there is at most one occurrence of an atom $\mathsf{R}(y_1, \ldots, y_{\#\mathsf{R}})$ in each rule from Δ. If this is not the case, we split the rule by introducing a new predicate symbol for each relation atom with relation symbol R_i, until the condition is satisfied.

▶ **Definition 13.** *An* unfolding tree *for a predicate atom $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$ is a Δ-labeled tree $\mathcal{T} = (\mathcal{N}, \mathcal{F}, r, \lambda)$, such that $\lambda(r)$ defines A and, for each vertex $n \in \mathcal{N}$, if $\mathsf{B}_1(z_{1,1}, \ldots, z_{1,\#\mathsf{B}_1})$, $\ldots$, $\mathsf{B}_h(z_{h,1}, \ldots, z_{h,\#\mathsf{B}_h})$ are the predicate atoms that occur in $\lambda(n)$, then $p_1, \ldots, p_h$ are the children of n in $\mathcal{T}$, such that $\lambda(p_\ell)$ defines B_ℓ, for all $\ell \in [1, h]$.*

We build a SO formula that defines the models of a relation atom $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$. As explained above, this is without loss of generality. Let P be the maximum number of occurrences of predicate atoms in a rule from Δ_ϕ. We use second order variables $Y_1, \ldots, Y_P$ of arity 2, for the edges of the unfolding tree and $X_1, \ldots, X_R$ of arity 1, for the labels of the nodes in the unfolding tree, i.e., the rules of Δ. First, we build a SO formula $\mathfrak{T}(x, \{X_i\}_{i=1}^{R}, \{Y_j\}_{j=1}^{P})$, as the conjunction of SO formulæ that describe the following facts:

- the root x belongs to X_i, for some rule r_i that defines A,
- the sets $X_1, \ldots, X_R$ are pairwise disjoint,
- each vertex in $X_1 \cup \ldots \cup X_R$ is reachable from x by a path with edges $Y_1, \ldots, Y_P$,
- each vertex in $X_1 \cup \ldots \cup X_R$, except for x, has exactly one incoming edge,
- x has no incoming edge,
- each vertex from X_i has exactly h outgoing edges $Y_1, \ldots, Y_h$, each to a vertex from X_{j_ℓ}, respectively, such that r_{j_ℓ} defines B_ℓ, for all $\ell \in [1,h]$, where $\mathsf{B}_1(z_{1,1}, \ldots, z_{1,\#\mathsf{B}_1}), \ldots, \mathsf{B}_h(z_{h,1}, \ldots, z_{h,\#\mathsf{B}_h})$ are the predicate atoms that occur in r_i.

Second, we build a SO formula expressing the relationship between the unfolding tree $\mathcal{T} = (\mathcal{N}, \mathcal{F}, r, \lambda)$ and the model. The formula $\mathfrak{F}(\xi_1, \ldots, \xi_{\#\mathsf{A}}, x, \{X_i\}_{i=1}^R, \{Y_j\}_{j=1}^P, \{\{Z_{k,\ell}\}_{\ell=1}^{\#\mathsf{R}_k}\}_{k=1}^N)$ uses second order variables $Z_{k,\ell}$, of arity 2, that encode partial functions mapping a tree node n to the value of ξ_ℓ for the (unique) atom $\mathsf{R}_k(\xi_1, \ldots, \xi_{\#\mathsf{R}_i})$ from the rule $\lambda(n)$, in case such an atom exists. The formula $\mathfrak{F}$ is the conjunction of following SO-definable facts[1]:

(i) each second order variable $Z_{k,\ell}$ denotes a functional binary relation,

(ii) for each tree node labeled by a rule r_i and each atom $\mathsf{R}_k(\xi_1, \ldots, \xi_{\#\mathsf{R}_k})$ occurring at that node, the interpretation of R_k contains a tuple, whose elements are related to the node via $Z_{k,1}, \ldots, Z_{k,\#\mathsf{R}_k}$, respectively,

(iii) for any (not necessarily distinct) rules r_i and r_j such that an atom with relation symbol R_k occurs in both, the corresponding tuples from the interpretation of R_k are distinct,

(iv) each tuple from the interpretation of R_k must have been introduced by a relation atom with relation symbol R_k that occurs in a rule r_i,

(v) two terms ξ_m and χ_n that occur in two relation atoms $\mathsf{R}_k(\xi_1, \ldots, \xi_{\#\mathsf{R}_k})$ and $\mathsf{R}_\ell(\chi_1, \ldots, \chi_{\#\mathsf{R}_\ell})$ within rules r_i and r_j, respectively, and are constrained to be equal (i.e., via equalities and parameter passing), must be equated,

(vi) a disequality $\xi \neq \chi$ that occurs in a rule r_i is propagated throughout the tree to each pair of variables that occur within two relation atoms $\mathsf{R}_k(\xi_1, \ldots, \xi_{\#\mathsf{R}_k})$ and $\mathsf{R}_\ell(\chi_1, \ldots, \chi_{\#\mathsf{R}_\ell})$ in rules r_{j_k} and r_{j_ℓ}, respectively, such that ξ is bound ξ_r and χ to χ_s by equalities and parameter passing,

(vii) each term in $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$ that is bound to a variable from a relation atom $\mathsf{R}_k(z_1, \ldots, z_{\#\mathsf{R}_k})$ in the unfolding, must be equated to that variable.

Summing up, the SO formula defining the models of the predicate atom $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$ with respect to the SID Δ is:

$$\begin{aligned}&\mathfrak{A}^{\mathsf{A}}_{\Delta}(\xi_1, \ldots, \xi_{\#\mathsf{A}}) \stackrel{\mathsf{def}}{=} \exists x \exists \{X_i\}_{i=1}^R \exists \{Y_j\}_{j=1}^P \exists \{Z_{1,\ell}\}_{\ell=1}^{\#\mathsf{R}_1} \ldots \exists \{Z_{K,\ell}\}_{\ell=1}^{\#\mathsf{R}_K} \,.\\ &\mathfrak{T}(x, \{X_i\}_{i=1}^R, \{Y_j\}_{j=1}^P) \wedge \ \mathfrak{F}(\xi_1, \ldots, \xi_{\#\mathsf{A}}, x, \{X_i\}_{i=1}^R, \{Y_j\}_{j=1}^P, \{\{Z_{k,\ell}\}_{\ell=1}^{\#\mathsf{R}_k}\}_{k=1}^N)\end{aligned}$$

The correctness of the above construction is proved in the following proposition, that also shows $[\![\mathsf{SLR}]\!] \subseteq [\![\mathsf{SO}]\!]$:

▶ **Proposition 14.** *Given an SID Δ and a predicate atom $\mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$, for each structure (U, σ) and store ν, we have $(U, \sigma) \models^\nu_\Delta \mathsf{A}(\xi_1, \ldots, \xi_{\#\mathsf{A}}) \Leftrightarrow (U, \sigma) \Vdash^\nu \mathfrak{A}^{\mathsf{A}}_{\Delta}(\xi_1, \ldots, \xi_{\#\mathsf{A}})$.*

We state as an open question whether the above formula can be written in ESO, which would sharpen the comparison between SLR and SO, as ESO is known to be strictly less expressive than SO [40]. In particular, the problem is writing $\mathfrak{F}$ in ESO.

[1] The exact SO formulæ are given in the full version of the paper [42].

6 $[\![\mathsf{MSO}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$

We prove that, for any MSO sentence ϕ and any integer $k \geq 1$, there exists an SID $\Delta(k,\phi)$ and a predicate $\mathsf{A}_{k,\phi}$ of arity zero, such that $[\![\phi]\!]^{\mathfrak{D},k} = [\![\mathsf{A}_{k,\phi}()]\!]_{\Delta(k,\phi)}$, i.e., the set of guarded models of ϕ of treewidth at most k corresponds to the set of structures SLR-defined by the predicate atom $\mathsf{A}_{k,\phi}()$, when interpreted in the SID $\Delta(k,\phi)$. Our proof leverages from a technique of Courcelle [23], used to show that the models of bounded treewidth of a given MSO sentence can be described by a finite set of recursive equations, written using an algebra of operations on structures. This result follows up in a long-standing line of work (known as Feferman-Vaught theorems [51]) that reduces the evaluation of an MSO sentence on the result of an algebraic operation to the evaluation of several related sentences in the arguments of the respective operation.

6.1 A Theorem of Courcelle

We recall first a result of Courcelle [23], that describes the structures of bounded treewidth, which satisfy a given MSO formula ϕ, by an effectively constructible set of recursive equations. This set of equations uses two operations on structures, namely *glue* and fgcst_j, that are lifted to sets of structures, as usual. The result is developed in two steps. The first step builds a generic set of equations, that characterizes all structures of treewidth at most k. This set of equations is then refined, in the second step, to describe only models of ϕ. Because this result applies to general (i.e., finite and infinite) structures (U,σ), we do not require U to be infinite, for the purposes of this presentation. We consider a fixed integer $k \geq 1$ and MSO sentence ϕ in the rest of this section.

Operations on Structures. Let Σ_1 and Σ_2 be two (possibly overlapping) signatures. The *glueing* operation $\mathit{glue} : \mathit{Str}(\Sigma_1) \times \mathit{Str}(\Sigma_2) \rightarrow \mathit{Str}(\Sigma_1 \cup \Sigma_2)$ is the union of structures with *disjoint universes*, followed by fusion of the elements denoted by constants. Formally, given $S_i = (U_i, \sigma_i)$, for $i = 1,2$, such that $U_1 \cap U_2 = \emptyset$, let $\sim$ be the least equivalence relation on $U_1 \cup U_2$ such that $\sigma_1(\mathsf{c}) \sim \sigma_2(\mathsf{c})$, for all $\mathsf{c} \in \Sigma_1 \cap \Sigma_2$. Let $[u]$ be the equivalence class of $u \in U_1 \cup U_2$ with respect to $\sim$ and lift this notation to tuples and sets of tuples. Then $\mathit{glue}(S_1,S_2) \stackrel{\text{def}}{=} (U,\sigma)$, where $U \stackrel{\text{def}}{=} \{[u] \mid u \in U_1 \cup U_2\}$ and σ is defined as follows:

$$\sigma(\mathsf{R}) \stackrel{\text{def}}{=} \begin{cases} [\sigma_i(\mathsf{R})], & \text{if } \mathsf{R} \in \Sigma_i \setminus \Sigma_{3-i}, \text{ for both } i = 1,2 \\ [\sigma_1(\mathsf{R}) \cup \sigma_2(\mathsf{R})], & \text{if } \mathsf{R} \in \Sigma_1 \cap \Sigma_2 \end{cases}$$

Since we match isomorphic structures, the nature of the elements of U (i.e., equivalence classes) is not important. The *forget* operation $\mathit{fgcst}_j : \mathit{Str}(\Sigma) \rightarrow \mathit{Str}(\Sigma \setminus \{\mathsf{c}_j\})$ simply drops the constant c_j from the domain of its argument.

Structures of Bounded Treewidth. Let $\Sigma = \{\mathsf{R}_1, \ldots, \mathsf{R}_N, \mathsf{c}_1, \ldots, \mathsf{c}_M\}$ be a signature and $\Pi = \{\mathsf{c}_{M+1}, \ldots, \mathsf{c}_{M+k+1}\}$ be a set of constants disjoint from Σ, called *ports*. We consider variables Y_i, for all subsets $\Pi_i \subseteq \Pi$, denoting sets of structures over the signature $\Sigma \cup \Pi_i$. The *equation system* $\mathit{Tw}(k)$ is the set of recursive equations of the form $Y_0 \supseteq f(Y_1, \ldots, Y_n)$, where each f is either *glue*, fgcst_{M+j}, for any $j \in [1,k+1]$, or a singleton relation of type R_i, consisting of a tuple with at most $k+1$ distinct elements, for any $i \in [1,N]$. It is known that the set of structures of treewidth at most k is a component of the least solution of $\mathit{Tw}(k)$, in the domain of tuples of sets ordered by pointwise inclusion [25, Theorem 2.83].

Models of MSO Formulæ. The *quantifier rank* $\mathrm{qr}(\phi)$ of an MSO formula ϕ is the maximal depth of nested quantifiers, i.e., $\mathrm{qr}(\phi) \stackrel{\text{def}}{=} 0$ if ϕ is an atom, $\mathrm{qr}(\neg\phi_1) \stackrel{\text{def}}{=} \mathrm{qr}(\phi_1)$, $\mathrm{qr}(\phi_1 \wedge \phi_2) \stackrel{\text{def}}{=} \max(\mathrm{qr}(\phi_1), \mathrm{qr}(\phi_2))$ and $\mathrm{qr}(\exists x \;.\; \phi_1) = \mathrm{qr}(\exists X \;.\; \phi_1) \stackrel{\text{def}}{=} \mathrm{qr}(\phi_1) + 1$. We denote by $\mathbb{F}^r_{\mathsf{MSO}}$ the set of MSO sentences of quantifier rank at most r. This set is finite, up to logical equivalence. For a structure $S = (U, \sigma)$, we define its r-*type* as $\mathit{type}^r(S) \stackrel{\text{def}}{=} \{\phi \in \mathbb{F}^r_{\mathsf{MSO}} \mid S \Vdash \phi\}$. We assume the sentences in $\mathit{type}^r(S)$ to use the signature over which S is defined; this signature will be clear from the context in the following.

▶ **Definition 15.** *An operation $f : \mathit{Str}(\Sigma_1) \times \ldots \times \mathit{Str}(\Sigma_n) \rightarrow \mathit{Str}(\Sigma_{n+1})$ is (effectively) MSO*-compatible[2] *iff, for all structures $S_1, \ldots, S_n$, $\mathit{type}^r(f(S_1, \ldots, S_n))$ depends only on (and can be effectively computed from) $\mathit{type}^r(S_1), \ldots, \mathit{type}^r(S_n)$ by an* abstract operation *$f^\sharp : (\mathrm{pow}(\mathbb{F}^r_{\mathsf{MSO}}))^n \rightarrow \mathrm{pow}(\mathbb{F}^r_{\mathsf{MSO}})$.*

The result of Courcelle establishes that glueing and forgetting of constants are effectively MSO-compatible operations, with effectively computable abstract operations $\mathit{glue}^\sharp$ and $\mathit{fgcst}^\sharp_{M+i}$, for $i \in [1, k+1]$, see [23, Lemmas 3.2 and 3.3]. As a consequence, one can build from $\mathit{Tw}(k)$ a set of recursive equations $\mathit{Tw}^\sharp(k)$ of the form $Y_0^{\tau_0} = f(Y_1^{\tau_1}, \ldots, Y_n^{\tau_n})$, where $Y_0 = f(Y_1, \ldots, Y_n)$ is an equation from $\mathit{Tw}(k)$ and $\tau_0, \ldots, \tau_n$ are r-types such that $\tau_0 = f^\sharp(\tau_1, \ldots, \tau_n)$. Intuitively, each annotated variable Y^τ denotes the set of structures whose r-type is τ, from the Y-component of the least solution of $\mathit{Tw}(k)$. Given some formula ϕ with $\mathrm{qr}(\phi) = r$, the set of models of ϕ of treewidth at most k is the union of the Y^τ-components of the least solution of $\mathit{Tw}^\sharp(k)$, such that $\phi \in \tau$ [23, Theorem 3.6].

6.2 Encoding Types in SLR

We begin explaining the proof for $[\![\mathsf{MSO}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$. Instead of using the set of recursive equations $\mathit{Tw}(k)$ from the previous subsection, we give an SID $\Delta(k)$ that characterizes the guarded structures of bounded treewidth (Fig. 1a). We use the separating conjunction to simulate the glueing operation. The main problem is with the interpretation of the separating conjunction, as composition of structures with possibly overlapping universes (Def. 2), that cannot be glued directly. Our solution is to consider guarded structures (Def. 1), where the unary relation symbol $\mathfrak{D}$ is used to enforce disjointness of the arguments of the composition operation, in all but finitely many elements. Intuitively, $\mathfrak{D}$ "collects" the values assigned to the existentially quantified variables created by rule (2) of $\Delta(k)$ and the top-level rule (4) during the unraveling. This ensures that

(i) the variables of a predicate atom are mapped to pairwise distinct values and

(ii) the composition of two guarded structures is the same as glueing them.

Similar conditions have been used to define e.g., fragments of SL with nice computational properties, such as the *establishment* condition used to ensure decidability of entailments [36], or the *tightness* condition from [4, §5.2].

To alleviate the presentation, the SID $\Delta(k)$ defines only structures $(U, \sigma) \in \mathit{Str}(\Sigma, \mathfrak{D})$ with at least $k+1$ distinct elements in $\sigma(\mathfrak{D})$ (rule 4) and $\sigma(\mathsf{R}) \neq \emptyset$ for at least one relation symbol $\mathsf{R} \in \Sigma$ (rule 3). The cases of structures such that $\|\sigma(\mathfrak{D})\| \leq k$ or $\bigcup_{\mathsf{R} \in \Sigma} \sigma(\mathsf{R}) = \emptyset$ can be dealt with easily, by adding more rules to $\Delta(k)$. In the rest of this section we show that $\Delta(k)$ defines all structures of k-bounded treewidth (except for the mentioned corner cases).

The main property of $\Delta(k)$ is stated below:

▶ **Lemma 16.** *For any guarded structure $(U, \sigma) \in \mathit{Str}(\Sigma, \mathfrak{D})$, such that $\|\sigma(\mathfrak{D})\| \geq k+1$ and $\sigma(\mathsf{R}) \neq \emptyset$, for at least some $\mathsf{R} \in \Sigma$, we have $\mathrm{tw}(\sigma) \leq k$ iff $(U, \sigma) \models_{\Delta(k)} \mathsf{A}_k()$.*

[2] Also referred to as *smooth* operations in [51].

$$\begin{aligned}
&\mathsf{A}(x_1,\ldots,x_{k+1}) \leftarrow \mathsf{A}(x_1,\ldots,x_{k+1}) * \mathsf{A}(x_1,\ldots,x_{k+1}) && (1)\\
&\mathsf{A}(x_1,\ldots,x_{k+1}) \leftarrow \exists y \;.\; \mathfrak{D}(y) * \mathsf{A}(x_1,\ldots,x_{k+1})[x_i/y] \text{ for all } i \in [1,k+1] && (2)\\
&\mathsf{A}(x_1,\ldots,x_{k+1}) \leftarrow \mathsf{R}(y_1,\ldots,y_{\#\mathsf{R}}) \text{ for all } \mathsf{R} \in \Sigma \text{ and } y_1,\ldots,y_{\#\mathsf{R}} \in \{x_1,\ldots,x_{k+1}\} && (3)\\
&\mathsf{A}_k() \leftarrow \exists x_1 \ldots \exists x_{k+1} \;.\; \mathfrak{D}(x_1) * \ldots * \mathfrak{D}(x_{k+1}) * \mathsf{A}(x_1,\ldots,x_{k+1}) && (4)
\end{aligned}$$

(a)

$$\begin{aligned}
&\mathsf{A}^{\tau}(x_1,\ldots,x_{k+1}) \leftarrow \mathsf{A}^{\tau_1}(x_1,\ldots,x_{k+1}) * \mathsf{A}^{\tau_2}(x_1,\ldots,x_{k+1}) \text{ where } \tau = \mathit{glue}^{\sharp}(\tau_1,\tau_2) && (5)\\
&\mathsf{A}^{\tau}(x_1,\ldots,x_{k+1}) \leftarrow \exists y \;.\; \mathfrak{D}(y) * \mathsf{A}^{\tau_1}(x_1,\ldots,x_{k+1})[x_i/y] \text{ for all } i \in [1,k+1], \text{where} && (6)\\
&\qquad \tau = \mathit{glue}^{\sharp}(\mathit{fgcst}^{\sharp}_{M+i}(\tau_1),\rho_i) \text{ and } \rho_i \text{ is the type of some structure}\\
&\qquad S \in \mathit{Str}(\{\mathsf{c}_{M+i}\},\mathfrak{D}) \text{ with singleton universe and } S \Vdash \mathfrak{D}(\mathsf{c}_{M+i})\\
&\mathsf{A}^{\tau}(x_1,\ldots,x_{k+1}) \leftarrow \mathsf{R}(y_1,\ldots,y_{\#\mathsf{R}}) \text{ for some } y_1,\ldots,y_{\#\mathsf{R}} \in \{x_1,\ldots,x_{k+1}\}, \text{ where} && (7)\\
&\qquad \tau = \mathit{type}^{\mathrm{qr}(\phi)}(S),\; S \in \mathit{Str}(\Sigma \cup \{\mathsf{c}_{M+1},\ldots,\mathsf{c}_{M+k+1}\},\mathfrak{D}) \text{ and}\\
&\qquad S \Vdash \mathsf{R}(y_1,\ldots,y_{\#\mathsf{R}})[x_1/\mathsf{c}_{M+1},\ldots,x_{k+1}/\mathsf{c}_{M+k+1}] * \mathop{\text{\LARGE $*$}}_{i=1}^{k+1}\mathfrak{D}(\mathsf{c}_{M+i})\\
&\mathsf{A}_{k,\phi}() \leftarrow \exists x_1 \ldots \exists x_{k+1} \;.\; \mathfrak{D}(x_1) * \ldots * \mathfrak{D}(x_{k+1}) \; * \mathsf{A}^{\tau}(x_1,\ldots,x_{k+1}) && (8)\\
&\qquad \text{for all } \tau \text{ such that } \phi \in \tau
\end{aligned}$$

(b)

Figure 1 The SID $\Delta(k)$ defining structures of treewidth at most k (a) and its annotation $\Delta(k,\phi)$ defining the models of an MSO sentence ϕ, of treewidth at most k (b).

We remark that the encoding of *glue* and fgcst_j used in the definition of $\Delta(k)$ can be used to show that any inductive set of structures, i.e., a set defined by finitely many recursive equations written using *glue* and fgcst_j, can be also defined in SLR. This means that SLR is at least as expressive than the inductive sets, which are always of bounded treewidth.

The second step of our construction is the annotation of the rules in $\Delta(k)$ with $\mathrm{qr}(\phi)$-types, in order to obtain an SID $\Delta(k,\phi)$ (Fig. 1b) describing the models of an MSO sentence ϕ, of treewidth at most k. We consider the set of ports $\Pi = \{\mathsf{c}_{M+1},\ldots,\mathsf{c}_{M+k+1}\}$ disjoint from Σ. The encoding of the store values of the variables $x_1,\ldots,x_{k+1}$ in a given structure is defined below:

▶ **Definition 17.** *Let $\Sigma = \{\mathsf{R}_1,\ldots,\mathsf{R}_N,\mathsf{c}_1,\ldots,\mathsf{c}_M\}$ be a signature, $\Pi = \{\mathsf{c}_{M+1},\ldots,\mathsf{c}_{M+k+1}\}$ be a set of constants not in Σ, and let $(U,\sigma) \in \mathit{Str}(\Sigma,\mathfrak{D})$ be a structure. Let ν be a store mapping $x_1,\ldots,x_{k+1}$ to elements of $U \setminus \sigma(\mathfrak{D})$. Then, $\mathit{encode}((U,\sigma),\nu) \in \mathit{Str}(\Sigma \cup \Pi,\mathfrak{D})$ is a structure with universe U that agrees with (U,σ) over Σ, maps each c_{M+i} to $\nu(x_i)$, for $i \in [1,k+1]$ and maps $\mathfrak{D}$ to $\sigma(\mathfrak{D}) \cup \{\nu(x_1),\ldots,\nu(x_{k+1})\}$.*

The correctness of our construction relies on the fact that the composition acts like glueing, for structures with universe U, whose sets of elements involved in the interpretation of some relation symbol may only overlap at the interpretation of the ports from Π:

▶ **Lemma 18.** *For an integer $r \geq 0$, a store ν and locally disjoint compatible structures $(U_1,\sigma_1),(U_2,\sigma_2) \in \mathit{Str}(\Sigma \cup \Pi,\mathfrak{D})$, such that $\mathrm{Rel}(\sigma_1) \cap \mathrm{Rel}(\sigma_2) \subseteq \{\sigma_1(\mathsf{c}_{M+1}),\ldots,\sigma_1(\mathsf{c}_{M+k+1})\}$ and $(\sigma_1(\mathfrak{D}) \cup \sigma_2(\mathfrak{D})) \cap \{\nu(x_i) \mid i \in [1,m]\} = \emptyset$, we have:*

$$\mathit{type}^{r}(\mathit{encode}((U_1,\sigma_1) \bullet (U_2,\sigma_2),\nu)) = \mathit{glue}^{\sharp}(\mathit{type}^{r}(\mathit{encode}((U_1,\sigma_1),\nu)),\mathit{type}^{r}(\mathit{encode}((U_2,\sigma_2),\nu)))$$

Finally, the main property of $\Delta(k,\phi)$ is stated and proved below:

▶ **Proposition 19.** *For any $k \geq 1$, MSO sentence ϕ, and guarded structure $(U,\sigma) \in Str(\Sigma,\mathfrak{D})$, the following are equivalent:*

(1) $(U,\sigma) \Vdash \phi$ *and* $\mathrm{tw}(\sigma) \leq k$*, and*

(2) $(U,\sigma) \models_{\Delta(k,\phi)} \mathsf{A}_{k,\phi}()$.

The above result shows that SLR can define the guarded models $(U,\sigma) \in Str(\Sigma,\mathfrak{D})$ of a given MSO formula whose treewidth is bounded by a given integer. We do not know, for the moment, if this result holds on unguarded structures as well.

The above construction of the SID $\Delta(k,\phi)$ is effectively computable, except for the rule (7), where one needs to determine the type of a structure $S = (U,\sigma)$ with infinite universe. However, we prove in the following that determining this type can be reduced to computing the type of a finite structure, which amounts to solving finitely many MSO model checking problems on finite structures, each of which being PSPACE-complete [59]. Given an integer $n \geq 0$ and a structure $S = (U,\sigma) \in Str(\Sigma)$, we define the finite structure $S^n = (\mathrm{Supp}(\sigma) \cup \{v_1,\ldots,v_n\},\sigma)$, for pairwise distinct elements $v_1,\ldots,v_n \in U \setminus \mathrm{Supp}(\sigma)$. Then, for any quantifier rank r, the structures S and S^{2^r} have the same r-type:

▶ **Lemma 20.** *Given $r \geq 0$ and $S = (U,\sigma) \in Str(\Sigma)$, we have $type^r(S) = type^r(S^{2^r})$.*

As a final remark, we notice that the idea used to prove $[\![\mathsf{MSO}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$ can be extended to show also $[\![\mathsf{CMSO}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$, where CMSO denotes the extension of MSO with cardinality constraints $\|X\|_{p,q}$ stating that the cardinality of a set of vertices X equals p modulo q, for some constants $0 \leq p < q$. This is because glueing and forgetting constants are CMSO-compatible operations [22, Lemma 4.5, 4.6 and 4.7].

7 The Remaining Cases

We discuss the results from Table 1, that are not already covered by §4, §5 and §6.

$[\![\mathsf{SO}]\!]^{\mathfrak{D},k} \not\subseteq [\![\mathsf{MSO}]\!]$. Since $[\![\mathsf{SLR}]\!] \subseteq [\![\mathsf{SO}]\!]$ and$[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \not\subseteq [\![\mathsf{MSO}]\!]$, we obtain that $[\![\mathsf{SO}]\!]^{\mathfrak{D},k} \not\subseteq [\![\mathsf{MSO}]\!]$. Moreover, $[\![\mathsf{SO}]\!] \not\subseteq [\![\mathsf{MSO}]\!]$ follows from the fact that our counterexample for $[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \not\subseteq [\![\mathsf{MSO}]\!]$ involves only structures of treewidth one.

$[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SO}]\!]$. By applying the translation of SLR to SO from §5 to $\Delta(k)$ (Fig. 1a) and to a given SID Δ defining a predicate A of zero arity, respectively, and taking the conjunction of the results with the SO formula defining guarded structures[3], we obtain an SO formula that defines the set $[\![\mathsf{A}()]\!]_{\Delta}^{\mathfrak{D},k}$, thus proving that $[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SO}]\!]$.

$[\![\mathsf{(M)SO}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{(M)SO}]\!]$. For each given $k \geq 1$, there exists an MSO formula θ_k that defines the structures of treewidth at most k [25, Proposition 5.11]. This is a consequence of the Graph Minor Theorem proved by Robertson and Seymour [55], combined with the fact that bounded treewidth graphs are closed under taking minors and that the property of having a given finite minor is MSO-definable[4]. Then, for any given (M)SO formula ϕ, the (M)SO formula $\phi \wedge \theta_k$ defines the models of ϕ of treewidth at most k.

[3] $\bigwedge_{\mathsf{R}\in\Sigma} \forall x_1 \ldots \forall x_{\#\mathsf{R}} \;.\; \mathsf{R}(x_1,\ldots,x_{\#\mathsf{R}}) \rightarrow \bigwedge_{i\in[1,\#\mathsf{R}]} \mathfrak{D}(x_i)$.

[4] The proof of Robertson and Seymour does not build θ_k, see [3] for an effective proof.

Open Problems. The following problems from Table 1 are currently open: $[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$ and $[\![\mathsf{SO}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$, both conjectured to have a negative answer. In particular, the difficulty concerning $[\![\mathsf{SLR}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$ is that, in order to ensure treewidth boundedness, it seems necessary to force the composition of structures to behave like glueing (see the definition of $\Delta(k)$ in Fig. 1a), which seems difficult without the additional relation symbol $\mathfrak{D}$.

Since $[\![\mathsf{MSO}]\!]^{\mathfrak{D},k} \subseteq [\![\mathsf{SLR}]\!]$ but $[\![\mathsf{MSO}]\!] \not\subseteq [\![\mathsf{SLR}]\!]$, we naturally ask for the existence of a fragment of SLR that describes only MSO-definable families of structures of bounded treewidth. In particular, [8, §6] defines a fragment of SLR that has bounded-treewidth models and is MSO-definable. However, in general, since SLR can define context-free sets of guarded graphs (the grammar in Figure 1a can be adapted to encode Hyperedge Replacement (HR) grammars [24]), the MSO-definability of a SLR-definable set is undecidable, as a consequence of the undecidability of the recognizability of context-free languages [39]. On the other hand, the treewidth-boundedness of a SLR-definable set is an open problem, that we conjecture decidable.

A possible direction for future work is also adding Boolean connectives to SLR. Here, one might study an SLR variant that supports Boolean connectives in a top-level logic but not within the inductive definitions, similar to the SL studied in [47, 53]. Adding Boolean connectives within the inductive definitions appears more difficult, as one will need to impose syntactic restitutions such as positive occurrences of predicate atoms in the right hand side of definitions or stratification of negation in order to ensure well-definedness.

8 Conclusions

We have compared the expressiveness of SLR, MSO and SO, in general and for models of bounded treewidth. Interestingly, we found that SLR and MSO are, in general, incomparable and subsumed by SO, whereas the models of bounded treewidth of MSO can be defined by SLR, modulo augmenting the signature with a unary relation symbol used to store the elements that occur in the original structure.

References

1 Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.

2 Peter Aczel. An introduction to inductive definitions. In *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. Elsevier, 1977. doi:10.1016/S0049-237X(08)71120-0.

3 Isolde Adler, Martin Grohe, and Stephan Kreutzer. Computing excluded minors. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 641–650, USA, 2008. Society for Industrial and Applied Mathematics.

4 Emma Ahrens, Marius Bozga, Radu Iosif, and Joost-Pieter Katoen. Reasoning about distributed reconfigurable systems. *Proc. ACM Program. Lang.*, 6(OOPSLA2):145–174, 2022. doi:10.1145/3563293.

5 Timos Antonopoulos and Anuj Dawar. Separating graph logic from mso. In Luca de Alfaro, editor, *Foundations of Software Science and Computational Structures*, pages 63–77, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

6 Mikołaj Bojańczyk and Michał Pilipczuk. Definability equals recognizability for graphs of bounded treewidth. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 407–416, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2933575.2934508.

7 Marius Bozga, Lucas Bueri, and Radu Iosif. Decision problems in a logic for reasoning about reconfigurable distributed systems. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning – 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 691–711. Springer, 2022. doi:10.1007/978-3-031-10769-6_40.

8 Marius Bozga, Lucas Bueri, and Radu Iosif. Decision problems in a logic for reasoning about reconfigurable distributed systems. *CoRR*, abs/2202.09637, 2022. arXiv:2202.09637.

9 Marius Bozga, Lucas Bueri, and Radu Iosif. On an invariance problem for parameterized concurrent systems. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.24.

10 Marius Bozga, Radu Iosif, and Joseph Sifakis. Verification of component-based systems with recursive architectures. *Theor. Comput. Sci.*, 940(Part):146–175, 2023. doi:10.1016/j.tcs.2022.10.022.

11 Rémi Brochenin, Stéphane Demri, and Étienne Lozes. On the almighty wand. *Inf. Comput.*, 211:106–137, 2012.

12 James Brotherston, Carsten Fuhs, Juan Antonio Navarro Pérez, and Nikos Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14–18, 2014*, pages 25:1–25:10. ACM, 2014. doi:10.1145/2603088.2603091.

13 James Brotherston, Nikos Gorogiannis, Max I. Kanovich, and Reuben Rowe. Model checking for symbolic-heap separation logic with inductive predicates. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016*, pages 84–96. ACM, 2016. doi:10.1145/2837614.2837621.

14 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), December 2011. doi:10.1145/2049697.2049700.

15 Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *Foundations of Software Science and Computational Structures*, pages 395–409, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

16 Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 366–378. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.30.

17 Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science, 21st Conference, Bangalore, India, December 13-15, 2001, Proceedings*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2001.

18 Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A Spatial Logic for Querying Graphs. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)*, volume 2380 of *Lecture Notes in Computer Science*, pages 597–610. Springer, July 2002. doi:10.1007/3-540-45465-9_51.

19 Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 365–377, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/325694.325742.

20 Matthew Collinson, Kevin McDonald, and David J. Pym. A substructural logic for layered graphs. *J. Log. Comput.*, 24(4):953–988, 2014. doi:10.1093/logcom/exu002.

21 Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory – 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2011. doi:10.1007/978-3-642-23217-6_16.

22 Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

23 Bruno Courcelle. The monadic second-order logic of graphs VII: graphs as relational structures. *Theor. Comput. Sci.*, 101(1):3–33, 1992. doi:10.1016/0304-3975(92)90148-9.

24 Bruno Courcelle. Monadic second-order definable graph transductions: A survey. *Theor. Comput. Sci.*, 126(1):53–75, 1994. doi:10.1016/0304-3975(94)90268-2.

25 Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2012. doi:10.1017/CBO9780511977619.

26 Anuj Dawar, Philippa Gardner, and Giorgio Ghelli. Expressiveness and Complexity of Graph Logic. *Information and Computation*, 205(3):263–310, February 2007. doi:10.1016/j.ic.2006.10.006.

27 Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors. *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science.* Springer, 2008. doi:10.1007/978-3-540-68679-8.

28 Stéphane Demri and Morgan Deters. Expressive completeness of separation logic with two variables and no separating conjunction. *ACM Trans. Comput. Log.*, 17(2):12, 2016.

29 Stéphane Demri, Étienne Lozes, and Alessio Mansutti. The effects of adding reachability predicates in quantifier-free separation logic. *ACM Trans. Comput. Log.*, 22(2):14:1–14:56, 2021. doi:10.1145/3448269.

30 Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics.* Springer, 2012.

31 Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 150–171, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

32 Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory.* Perspectives in Mathematical Logic. Springer, 1995.

33 Mnacho Echenim, Radu Iosif, and Nicolas Peltier. The bernays-schönfinkel-ramsey class of separation logic with uninterpreted predicates. *ACM Trans. Comput. Log.*, 21(3):19:1–19:46, 2020.

34 Mnacho Echenim, Radu Iosif, and Nicolas Peltier. Decidable Entailments in Separation Logic with Inductive Definitions: Beyond Establishment. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, volume 183 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2021.20.

35 Mnacho Echenim, Radu Iosif, and Nicolas Peltier. Unifying decidable entailments in separation logic with inductive definitions. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28 – 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 183–199. Springer, 2021. doi:10.1007/978-3-030-79876-5_11.

36 Mnacho Echenim, Radu Iosif, and Nicolas Peltier. Entailment is undecidable for symbolic heap separation logic formulæ with non-established inductive rules. *Inf. Process. Lett.*, 173:106169, 2022. doi:10.1016/j.ipl.2021.106169.

37 Diego Figueira and Leonid Libkin. Path logics for querying graphs: Combining expressiveness and efficiency. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 329–340. IEEE Computer Society, 2015. `doi:10.1109/LICS.2015.39`.

38 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. `doi:10.1007/3-540-29953-X`.

39 S. Greibach. A note on undecidable properties of formal languages. *Math. Systems Theory*, 2:1–6, 1968. `doi:10.1007/BF01691341`.

40 Neil Immerman. *Second-Order Logic and Fagin's Theorem*, pages 113–124. Springer New York, New York, NY, 1999. `doi:10.1007/978-1-4612-0539-5_8`.

41 Radu Iosif, Adam Rogalewicz, and Jirí Simácek. The tree width of separation logic with recursive definitions. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24 – 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2013. `doi:10.1007/978-3-642-38574-2_2`.

42 Radu Iosif and Florian Zuleger. Expressiveness results for an inductive logic of separated relations, 2023. `arXiv:2307.02381`.

43 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26. ACM, 2001. `doi:10.1145/360204.375719`.

44 Christina Jansen, Jens Katelaan, Christoph Matheja, Thomas Noll, and Florian Zuleger. Unified reasoning about robustness properties of symbolic-heap separation logic. In *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 611–638. Springer, 2017.

45 Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 66–74, New York, NY, USA, 1982. Association for Computing Machinery. `doi:10.1145/582153.582161`.

46 Jens Katelaan, Dejan Jovanovic, and Georg Weissenbacher. A separation logic with data: Small models and automation. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning – 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 455–471. Springer, 2018. `doi:10.1007/978-3-319-94205-6_30`.

47 Jens Katelaan and Florian Zuleger. Beyond symbolic heaps: Deciding separation logic with inductive definitions. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 390–408. EasyChair, 2020. `doi:10.29007/vkmj`.

48 Viktor Kuncak and Martin Rinard. Generalized records and spatial conjunction in role logic. In *Static Analysis*, pages 361–376, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

49 Viktor Kuncak and Martin C. Rinard. On spatial conjunction as second-order logic. *CoRR*, cs.LO/0410073, 2004. URL: `http://arxiv.org/abs/cs.LO/0410073`.

50 Étienne Lozes. *Expressivité des logiques spatiales*. Thèse de doctorat, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, France, November 2004. URL: `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/PhD-lozes.ps`.

51 Johann A. Makowsky. Algorithmic uses of the feferman-vaught theorem. *Ann. Pure Appl. Log.*, 126(1-3):159–213, 2004.

52 Alessio Mansutti. *Logiques de séparation : complexité, expressivité, calculs. (Reasoning with separation logics : complexity, expressive power, proof systems)*. PhD thesis, University of Paris-Saclay, France, 2020. URL: `https://tel.archives-ouvertes.fr/tel-03094373`.

53 Christoph Matheja, Jens Pagel, and Florian Zuleger. A decision procedure for guarded separation logic complete entailment checking for separation logic with inductive definitions. *ACM Trans. Comput. Log.*, 24(1):1:1–1:76, 2023. doi:10.1145/3534927.

54 Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999.

55 M. R. Fellows R. G. Downey. *Parameterized Complexity.* Springer New York, NY, 1999. doi:10.1007/978-1-4612-0515-9.

56 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

57 D. Seese. The structure of the models of decidable monadic theories of graphs. *Annals of Pure and Applied Logic*, 53(2):169–195, 1991. doi:10.1016/0168-0072(91)90054-P.

58 Dirk van Dalen. *Logic and structure (3. ed.).* Universitext. Springer, 1994.

59 Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982.

Hypernode Automata

Ezio Bartocci ✉ 🏠
Technische Universität Wien, Austria

Thomas A. Henzinger ✉ 🏠
IST Austria, Klosterneuburg, Austria

Dejan Nickovic ✉
AIT Austrian Institute of Technology, Wien, Austria

Ana Oliveira da Costa ✉
IST Austria, Klosterneuburg, Austria

Abstract

We introduce *hypernode automata* as a new specification formalism for hyperproperties of concurrent systems. They are finite automata with nodes labeled with *hypernode logic* formulas and transitions labeled with actions. A hypernode logic formula specifies relations between sequences of variable values in different system executions. Unlike HyperLTL, hypernode logic takes an *asynchronous* view on execution traces by constraining the values and the order of value changes of each variable without correlating the timing of the changes. Different execution traces are synchronized solely through the transitions of hypernode automata. Hypernode automata naturally combine asynchronicity at the node level with synchronicity at the transition level. We show that the model-checking problem for hypernode automata is decidable over action-labeled Kripke structures, whose actions induce transitions of the specification automata. For this reason, hypernode automaton is a suitable formalism for specifying and verifying asynchronous hyperproperties, such as declassifying observational determinism in multi-threaded programs.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Hyperproperties, Asynchronous, Automata, Logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.21

Related Version *Full Version*: `https://arxiv.org/abs/2305.02836`

Funding This work was supported in part by the Austrian Science Fund (FWF) SFB project SpyCoDe F8502, by the FWF projects ZK-35 and W1255-N23, and by the ERC Advanced Grant VAMOS 101020093.

1 Introduction

Formalisms like Linear temporal logic (LTL) or automata are commonly used to specify and verify trace properties of concurrent systems. Security requirements such as information-flow policies require simultaneous reasoning about multiple execution traces; hence, they cannot be expressed as trace properties. Hyperproperties address this limitation by specifying properties of trace sets [11]. HyperLTL [10], an extension of LTL with trace quantifiers, has emerged as a popular formalism for both the specification and verification of an important class of hyperproperties. The temporal operators of HyperLTL and related hyperlogics progress in lockstep over all traces that are bound to a trace variable; they specify *synchronous* hyperproperties. As a consequence, HyperLTL cannot specify, for instance, an information-flow policy that changes from one system mode to another if the mode transition can occur at different times in different system executions [3]. This limitation has been observed repeatedly and independently in recent years by [12, 8, 5] all of whom have proposed asynchronous versions of hyperlogics to address the problem.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 21; pp. 21:1–21:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We take a different route and propose a specification language for hyperproperties, called *hypernode automata*, which combines synchronicity and asynchronicity by combining automata and logic. *Hypernode automata* are finite automata with nodes labeled with formulas from a fully asynchronous, non-temporal hyperlogic, called *hypernode logic*, and transitions labeled with actions used to synchronize different execution traces. While automata-based languages have been used before for specifying synchronous hyperproperties [7], hypernode automata are the first language that systematically separates trace synchronicity from trace asynchronicity in the specification of hyperproperties: within hypernodes (i.e., states of the hypernode automata), different execution traces proceed at independent speeds, only to "wait for each other" when they transition to the next hypernode. This separation leads to natural specifications and plays to the strengths of both automata-based and logic-based formalisms.

Hypernodes' specification adopts a maximally asynchronous view over finite trace segments: each program variable can progress independently. We introduce *hypernode logic* to specify such asynchronous hyperproperties. Hypernode logic includes quantification over finite traces and the binary relation $x(\pi) \precsim y(\pi')$, for trace variables π and π', and system (or program) variables x and y. This relation asserts that the program variable x undergoes the same ordered value changes in the trace assigned to π as the variable y does in π', but the changes may happen at different times (stuttering), and there may be additional changes of y in π' (prefixing). The *stutter-reduced prefixing* relation $\precsim$ between finite traces is the only nonlogical operator of hypernode logic, yielding a novel, elegant, and powerful method to specify asynchronous hyperproperties over finite traces.

For each finite or infinite action sequence, a *hypernode automaton* specifies a corresponding sequence of formulas from hypernode logic. This paper's main contribution is a model-checking algorithm for hypernode automata. Our algorithm checks if a given hypernode automaton accepts the set of (possibly) infinite traces defined by an action-labeled Kripke structure. The action labels on transitions of the Kripke structure (the "model") induce equally labeled transitions of the hypernode automaton (the "specification"). The subroutine that model-checks formulas of hypernode logic is technically novel: it introduces automata-theoretic constructions on a new concept called *stutter-free automata*, which are then used in familiar logical contexts such as filtration and self-composition. While hypernode logic has existential trace quantifiers and thus can specify nonsafety hyperproperties such as the independence of variables [3], we focus in this paper on safe hypernode automata to specify hyperproperties of infinite executions. To our knowledge, this is the first decidability result for a simple but general formalism that can specify important asynchronous hyperproperties.

Perhaps the most famous asynchronous hyperproperty is *observational determinism* [17]. In Section 2, we motivate hypernode logic by providing a formal specification of observational determinism as defined by [17]. We further motivate hypernode automata by specifying *declassification* of information, which can be represented by a transition between hypernodes. Section 3 defines hypernode logic and hypernode automata. In Section 4, we first solve the model-checking problem for hypernode logic over Kripke structures using stutter-reduced automata, and then the more general model-checking problem for hypernode automata over action-labeled Kripke structures. In contrast to previously proposed specification formalisms for asynchronous hyperproperties [12, 8, 5], which are undecidable in general and decidable only for specific fragments, our model-checking algorithms are doubly exponential in the number of variables in the Kripke structure. Finally, in Section 5, we argue that our formalism is expressively incomparable to these other formalisms. For this reason, hypernode automaton is a promising specification formalism for asynchronous hyperproperties and thus significantly contributes to the automatic verification of security properties.

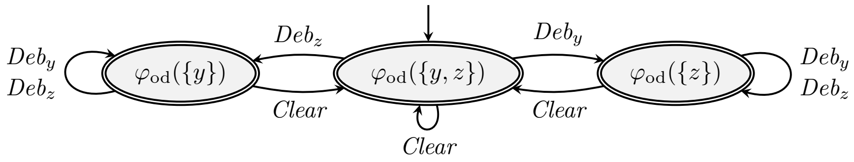

Deb_y
Deb_z
$\varphi_{\mathrm{od}}(\{y\})$
Deb_z
$Clear$
$\varphi_{\mathrm{od}}(\{y,z\})$
$Clear$
Deb_y
$Clear$
$\varphi_{\mathrm{od}}(\{z\})$
Deb_y
Deb_z

Figure 1 Hypernode automaton $\mathcal{H}$ specifying the mutually exclusive declassification of secure information, where $\varphi_{\mathrm{od}}(L) \overset{\mathrm{def}}{=} \forall\pi\forall\pi' \bigwedge_{l\in L}(l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi))$.

2 Motivating Example

The seminal work by Zdancewic and Myers [17] proposes the notion of observational determinism to specify information-flow policies of concurrent programs. Observational determinism is a noninterference property which requires publicly visible values to not depend on secret information. Noninterference specification is particularly challenging for multi-threaded programs because (i) the executions of a multi-threaded program depend on the scheduling policy, and (ii) a change from one program state to another can happen at different times in each execution of the program. According to [17], a program is observationally deterministic if, when starting from any two low-equivalent states, then any two traces of each low variable are equivalent up to *stuttering* and *prefixing*. This definition takes an asynchronous view of execution traces, so HyperLTL is not adequate to specify it [3].

In this section, we use hypernode logic to specify *Zdancewic-Myers observational determinism*, and use it to define a *mutually exclusive declassification policy* with a hypernode automaton. The declassification policy involves not only the asynchronous requirement of observational determinism, but also dynamic changes between different observational determinism requirements. In particular, the policy requires that during *normal* operation, two publicly visible variables y and z must not leak secret information. The policy also admits two *debugging* modes, in which either y or z can leak information, but never both. The "mode operation" is inspired by examples on declassification policies by the programming languages community (c.f. [2, 15]).

The hypernode automaton specification $\mathcal{H}$ of the declassification policy is shown in Figure 1. A hypernode automaton is interpreted over a set of action-labeled traces, which are sequences of valuations for program variables and actions. The transitions between automaton nodes are labeled with actions marking when the program changes its mode of operation, say, from normal mode to one of the two debugging modes. In the example, transitions from the normal mode to the two debugging modes are labeled with Deb_y and Deb_z actions, respectively; transitions from either debugging mode to the normal mode are labeled with *Clear*.

The automaton nodes are labelled with formulas of hypernode logic. In our example, all three formulas specify Zdancewic-Myers observational determinism but for different program variables. Observational determinism requires that for any two program executions (specified by $\forall\pi\forall\pi'$), their projections to each publicly visible program variable (l in a set L of public variables) are equivalent up to stuttering and prefixing (specified by $l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi)$). The visible program variables change from mode to mode, so the different specification nodes are labeled with different instances of the same formula. For example, the hypernode with formula $\varphi_{\mathrm{od}}(\{y\})$ requires observational determinism only for y.

Algorithm 1 defines a reactive program $\mathcal{P}_{\mathrm{var}}$ (where var can be either y or z) which in every iteration reads the input variable x and the action status. If, for example, the action is Deb_y, then variable y is used for debugging and the program copies the content of x to y.

Algorithm 1 Program $\mathcal{P}_{\text{var}}$.

```
1 do
2     var := 0;
3     read($x$);
4     read(status);
5     if (status = $Deb_{\text{var}}$) then
6         var := $x$;
7     end
8     output(var);
9 while true;
```

Table 1 Executions of $\mathcal{P}_y \parallel \mathcal{P}_z$.

τ_1	x:	0	0	0	
	y:	0	0	0	
	z:	0	0	0	
	status:	ε	Deb_y	Deb_z	
τ_2	x:	1	1	1	1
	y:	0	0	1	1
	z:	0	0	0	1
	status:	ε	ε	Deb_y	Deb_z

The parallel composition $\mathcal{P}_y \parallel \mathcal{P}_z$ does not satisfy the specification $\mathcal{H}$. Consider, for instance, the set $T = \{\tau_1, \tau_2\}$ of traces shown in Table 1. We make two important observations on τ_1 and τ_2: (1) these traces have different lengths, and (2) they exhibit the *same* sequence of actions (Deb_y followed by Deb_z) happening at *different* times (hence the traces are asynchronous). We note that the above sequence of actions partitions each trace into a sequence of three trace segments, called *slices*, which we denote using the white, the light gray, and the dark gray color in Table 1. We map each slice to a unique node in the hypernode automaton. The white slice of T is mapped to the initial node of $\mathcal{H}$. The light-gray slice of T is mapped to the node accessible from the initial state with action Deb_y (i.e., the debugging mode for y), which is labeled by the formula $\varphi_{\text{od}}(\{z\})$. The dark-gray slice of T is mapped to the same node, because the action Deb_z triggers the self-loop transition.

A sequence of actions defines a path in the hypernode automaton. Then, a set T of traces with a sequence of actions p satisfies the hypernode automaton $\mathcal{H}$ iff the slicing of T induced by p satisfies the hypernode formulas in the path defined by p in $\mathcal{H}$. This is not the case in our example because the dark-gray slice of T violates its associated hypernode formula $\varphi_{\text{od}}(\{z\})$. More specifically, the program variable z evaluates to 0 in the dark-gray segment of the trace τ_1, while it evaluates to 1 in the dark-gray segment of τ_2. The specification violation occurs because the critical section (lines $5 - 7$ in Algorithm 1) is unprotected. It is possible that the action Deb_z happens (line 4 in $\mathcal{P}_z$) after Deb_y. Thereafter the input value x is copied to z (line 6 in $\mathcal{P}_z$), and both y and z are made observable (line 8 in both $\mathcal{P}_y$ and $\mathcal{P}_z$). Hence they both leak information about x, which violates the specification. Our model-checking algorithm allows us to fully automate the reasoning in this example.

3 Hypernode Automata

In this section, we define *hypernode logic* and *hypernode automata*. We represent program executions as finite or infinite sequences of finite trace segments with synchronization actions. Let X be a finite set of *program variables* over a finite domain Σ, A be a finite set of *actions* and $A_\varepsilon = A \cup \{\varepsilon\}$.

Hypernode logic is interpreted over finite trace segments. A *trace segment* τ is a finite sequence of valuations in Σ^X, where each *valuation* $v\colon X \to \Sigma$ maps program variables to domain values. We denote the set of trace segments over X and Σ by $(\Sigma^X)^*$. A *segment property* T is a set of trace segments, that is, $T \subseteq (\Sigma^X)^*$. A formula of hypernode logic specifies a property of a set of trace segments, which is called a *segment hyperproperty*. Formally, a segment hyperproperty $\mathbf{T}$ is a set of segment properties, that is, $\mathbf{T} \subseteq 2^{(\Sigma^X)^*}$.

Hypernode automata are interpreted over finite and infinite action-labeled traces. An *action-labeled trace* ρ is a finite or infinite sequence of pairs, each consisting of a valuation and either an action label from A, or the empty label ε; that is, $\rho \in (\Sigma^X \times A_\varepsilon)^*$ or $\rho \in (\Sigma^X \times A_\varepsilon)^\omega$.

We require, for technical simplicity, that for infinite action-labeled traces, infinitely many labels are non-empty. An *action-labeled trace property* is a set of action-labeled traces. A hypernode automaton accepts action-labeled trace properties, and thus specifies an *action-labeled trace hyperproperty*, namely, the set of all action-labeled trace properties it accepts.

3.1 Hypernode Logic

Hypernode logic, FO[$\precsim$], is a first-order formalism to specify relations between the changes of the values of program variables over a set of trace segments. The formulas of hypernode logic are defined by the grammar: $\varphi \ ::= \ \exists\pi\,\varphi \,|\, \neg\varphi \,|\, \varphi \wedge \varphi \,|\, x(\pi) \precsim x(\pi)$, where the first-order variable π ranges over the set $\mathcal{V}$ of trace variables and the unary function symbol x ranges over the set X of program variables. Hypernode logic refers to time only through the binary *stutter-reduced prefixing* predicate $\precsim$. The intended meaning of the atomic formula $x(\pi) \precsim y(\pi')$ is that x undergoes the same ordered value changes in the trace segment assigned to π as the variable y does in π', followed by possibly additional value changes of y in π'. In other words, hypernode logic adopts a fully asynchronous comparison of different trace segments in which all variables are considered separately.

We therefore interpret hypernode formulas over *unzipped* trace segments, which encode the evolution of each program variable independently. An *unzipped trace segment* $\tau : X \to \Sigma^*$ is a function from the program variables to finite strings of values. The formulas of hypernode logic are interpreted over assignments of trace variables to unzipped trace segments. Given a set $T \subseteq (\Sigma^*)^X$ of unzipped trace segments, an assignment $\Pi_T : \mathcal{V} \to T$ maps each trace variable to an unzipped trace segment in T. We denote by $\Pi_T[\pi \mapsto \tau]$ the update of Π_T, where π is assigned to τ. The satisfaction relation for a formula φ of hypernode logic over an assignment Π_T is defined inductively as follows:

$$\begin{aligned}
&\Pi_T \models \exists\pi\varphi \text{ iff } \text{ there exists } \tau \in T :\ \Pi_T[\pi \mapsto \tau] \models \varphi;\\
&\Pi_T \models \psi_1 \wedge \psi_2 \text{ iff } \Pi_T \models \psi_1 \text{ and } \Pi_T \models \psi_2;\quad \Pi_T \models \neg\psi_1 \text{ iff } \Pi_T \not\models \psi_1;\\
&\Pi_T \models x(\pi) \precsim y(\pi') \text{ iff } \Pi_T(\pi)(x) \in \sigma_0^+ \ldots \sigma_n^+ \text{ and } \Pi_T(\pi')(y) \in \sigma_0^+ \ldots \sigma_n^+ \Sigma^*\\
&\qquad\qquad\qquad\qquad\qquad \text{with } \sigma_i \neq \sigma_{i+1},\ \text{for } 0 \leq i < n.
\end{aligned}$$

A set T of unzipped trace segments is a *model* of the formula φ, denoted by $T \models \varphi$, iff there exists an assignment Π_T such that $\Pi_T \models \varphi$. We adopt the usual abbreviations $\forall\pi\varphi \overset{\text{def}}{=} \neg\exists\pi\neg\varphi$ and $\varphi \vee \varphi' \overset{\text{def}}{=} \neg(\neg\varphi \wedge \neg\varphi')$. From now on, unless stated otherwise, program and trace variables are indexed by a natural number, i.e., $X = \{x_1, \ldots, x_m\}$ and $\mathcal{V} = \{\pi_1, \ldots, \pi_n\}$.

▶ **Example 1.** We illustrate how to use hypernode logic by specifying four different variants of non-interference between program variables. Zdancewic and Myers introduced in [17] the first notion of observational determinism to capture non-interference for concurrent programs. They require that in every program execution, every publicly visible variable in a set L must be stutter-equivalent up to prefixing (i.e., one of the executions can have more value changes): $\forall\pi\forall\pi' \bigwedge_{l\in L} (l(\pi) \precsim l(\pi') \vee l(\pi') \precsim l(\pi))$. Later, Huisman, Worah, and Sunesen [13] strengthened the previous definition by requiring every publicly visible variable to be stutter-equivalent in all executions: $\forall\pi\forall\pi' \bigwedge_{l\in L} l(\pi) \precsim l(\pi')$. Our third variant of observational determinism is from Terauchi[16], requiring the set of all publicly visible variables to be stutter-equivalent up to prefixing: $\forall\pi\forall\pi'\ (L(\pi) \precsim L(\pi') \vee L(\pi') \precsim L(\pi))$. Note that, we can encode the values of a finite set of variables within a single variable called L because we interpreted hypernode formulas over arbitrary finite domains. Finally, we specify independence (also known as generalized non-interference [11]) as defined in [3]. Two program variables x and y are *independent* iff whenever a sequence of value changes for x is possible in some trace π, and a sequence of value changes for y is possible in some trace π', then also their combination $(x(\pi), y(\pi'))$ is possible in some trace. The formula for independence specifies that for every two traces (π and π') there exists a third trace ($\pi_\exists$) that witnesses the combination $(x(\pi), y(\pi'))$ up to stuttering and prefixing: $\forall\pi\forall\pi'\exists\pi_\exists\ (x(\pi) \precsim x(\pi_\exists) \wedge y(\pi') \precsim y(\pi_\exists))$. $\lhd$

Stutter-reduced trace segments

We are interested in unzipped trace segments that are stutter-free, i.e., that do not repeat the same variable value in consecutive time points. For a program variable x and unzipped trace segment τ with $\tau(x) \in \sigma_0^+ \dots \sigma_n^+$ where $\sigma_i \neq \sigma_{i+1}$ for $i < n$, the *stutter-reduction* is $\lfloor \tau(x) \rfloor = \sigma_0 \dots \sigma_n$. We extend this notion naturally to the stutter-reduction of τ by $\lfloor \tau \rfloor(x) = \lfloor \tau(x) \rfloor$ for all program variables $x \in X$, and to the stutter reduction of a set T of unzipped trace segments by $\lfloor T \rfloor = \{\lfloor \tau \rfloor \mid \tau \in T\}$. We prove that formulas of hypernode logic cannot distinguish between a set of unzipped trace segments T and its stutter-reduction $\lfloor T \rfloor$.

▶ **Proposition 2.** *Let $T \subseteq (\Sigma^*)^X$ be a set of unzipped trace segments and φ a formula of hypernode logic. Then, $T \models \varphi$ iff $\lfloor T \rfloor \models \varphi$.*

3.2 Hypernode Automata

Hypernode automata are finite automata with states (called *hypernodes*) labeled with formulas of hypernode logic and transitions labeled with actions. A hypernode automaton reads a set $R \subseteq (\Sigma^X \times A_\varepsilon)^\omega$ of action-labeled traces, and accepts some of these sets.

▶ **Definition 3.** *A deterministic, finite* hypernode automaton *(HNA) is a tuple $\mathcal{H} = (Q, \hat{q}, \gamma, \delta)$, where Q is a finite set of states with $\hat{q} \in Q$ being the initial state, the state labeling function γ assigns a closed formula of hypernode logic over the program variables X to each state in Q, and the transition function $\delta : Q \times A \to Q$ is a total function assigning to each state and action a unique successor state.*

We assume the totality and determinism of the transition function only for the simplicity of the technical presentation. A *run* of the HNA $\mathcal{H}$ is a finite or infinite sequence $r = q_0 a_0\, q_1 a_1\, q_2 a_2 \dots$ of alternating hypernodes and actions which starts in the initial hypernode $q_0 = \hat{q}$ and follows the transition function, i.e., $\delta(q_i, a_i) = q_{i+1}$ for all $i \geq 0$. We refer to the corresponding sequence $p = a_0 a_1 a_2 \dots$ of actions as the *action sequence* of r. Note that each action sequence defines a unique run of $\mathcal{H}$.

The *action sequence* of an action-labeled trace $\rho = (v_0, a_0)(v_1, a_1)(v_2, a_2)\dots$, where $v_i \in \Sigma^X$ and $a_i \in A_\varepsilon$ for all $i \geq 0$, is the projection of the trace to its actions, with all empty labels ε removed; that is, $\rho[A] = a_0' a_1' \dots$ with $a_0 a_1 \dots \in a_0' \varepsilon^* a_1' \varepsilon^* \dots$ and $a_i' \in A$ for all $i \geq 0$. Given a set R of action-labeled traces, the *projection of R with respect to a finite action sequence* $p \in A^*$ is $R[p] = \{\rho \in R \mid \rho[A] = p p'$ for some suffix $p' \in A^* \cup A^\omega\}$.

Each step in the run of a hypernode automaton defines a new *slice* on a set of action-labeled traces. Let $p = a_0 a_1 \dots a_n$ be a finite action sequence, and $\rho = (v_0, a_0')(v_1, a_1')\dots$ be an action-labeled trace that has prefix p, and let R be a set of such traces. We write $\rho(\varnothing, a_0)$ for the initial trace segment of ρ which ends with the action label a_0. Formally, $\rho(\varnothing, a_0) = v_0 \dots v_k$ such that $a_k' = a_0$, and $a_i' = \varepsilon$ for all $0 \leq i < k$. Furthermore, we write $\rho(a_0 a_1 \dots a_i, a_{i+1})$ for the subsequent trace segments of ρ which end with the action label a_{i+1} after having seen the action sequence $a_0 a_1 \dots a_i$. Inductively, if $\rho(a_0 a_1 \dots a_{i-1}, a_i) = v_k \dots v_l$, $a_m' = a_{i+1}$ for $m > l$, and $a_j' = \varepsilon$ for all $l < j < m$, then $\rho(a_0 a_1 \dots a_i, a_{i+1}) = v_{l+1} \dots v_m$. The slicing is extended to sets of action-labeled traces accordingly; for example, $R(\varnothing, a) = \{\rho(\varnothing, a) \mid \rho \in R\}$.

Since the formulas of hypernode logic are interpreted over unzipped trace segments, in a final step, we need to unzip each slice. The *unzipping of a trace segment* $\tau = v_0 \dots v_n$ over the set of variables $X = \{x_1, \dots, x_m\}$ is $\text{unzip}(\tau) = \{x_0 : v_0(x_0)..v_n(x_0), \dots, x_m : v_0(x_m)..v_n(x_m)\}$. We define the unzipping of trace segments sets naturally as $\text{Unzip}(T) = \{\text{unzip}(\tau) \mid \tau \in T\}$.

▶ **Definition 4.** *Let $\mathcal{H} = (Q, \hat{q}, \gamma, \delta)$ be an HNA, and R a set of action-labeled traces. Let p be a finite action sequence in A^*. The set R is* accepted *by $\mathcal{H}$ with respect to the pattern p, denoted $R \models_p \mathcal{H}$, iff for the run $\mathcal{H}[p] = q_0 a_0\, q_1 a_1 \,\dots\, q_n a_n$, all slices of R induced by p are models of the formulas that label the respective hypernodes; that is,* $\text{Unzip}(R[p](\varnothing, a_0)) \models \gamma(q_0)$*, and* $\text{Unzip}(R[p](a_0 \dots a_{i-1}, a_i)) \models \gamma(q_i)$ *for all $0 < i \leq n$.*

A set R of action-labeled traces is *accepted* by the HNA $\mathcal{H}$ iff for all finite action sequences $p \in A^*$, if $R[p] \neq \emptyset$, then $R \models_p \mathcal{H}$. The *language* accepted by $\mathcal{H}$ is the set of all sets of action-labeled traces that are accepted by $\mathcal{H}$, denoted $\mathcal{L}(\mathcal{H})$. Note that this definition assumes that all finite and infinite runs of HNA are feasible; such automata are often called *safety automata*. Refinements are possible where finite runs must end in accepting states or, for example, infinite runs must visit accepting states infinitely often.

4 Model Checking

We present an algorithm for the model-checking problem for hypernode automata over Kripke structures whose transitions are labeled with actions.

4.1 Action-labeled Kripke Structures

A *Kripke structure* is a tuple $K = (W, \Sigma^X, \Delta, V)$ consisting of a finite set W of worlds, a set X of variables over a finite domain Σ, a transition relation $\Delta \subseteq W \times W$, and a value assignment $V : W \times X \to \Sigma$ that assigns a value from the finite domain Σ to each variable in each world. Given a Kripke structure with a transition relation Δ, and given a set A of actions, an *action labeling* for K over A is a function $\mathbb{A} : \Delta \to 2^{A_\varepsilon}$ that assigns a set of action labels (including possibly the empty label ε) to each transition. A *pointed Kripke structure* is a Kripke structure with one of its worlds being an initial world, denoted (K, w_0) with $w_0 \in W$.

A *path* in the Kripke structure K with action labeling $\mathbb{A}$ is a finite or infinite sequence $w_0 a_0\, w_1 a_1\, w_2 a_2 \dots$ of alternating worlds and actions which respects both the transition relation, $(w_i, w_{i+1}) \in \Delta$, and the action labeling, $a_i \in \mathbb{A}(w_i, w_{i+1})$, for all $i \geq 0$. We write $\text{Paths}(K, \mathbb{A})$ for the set of all such paths. The path $\varrho = w_0 a_0\, w_1 a_1 \dots$ defines the action-labeled trace $\text{zip}(\varrho) = V(w_0) a_0\, V(w_1) a_1 \dots$. We write $\text{Zip}(K, \mathbb{A})$ for the set of action-labeled traces defined by paths in $\text{Paths}(K, \mathbb{A})$. By $\text{Paths}(K, \mathbb{A}, w_0)$ we denote the set of all paths in $\text{Paths}(K, \mathbb{A})$ that start at the world w_0. As before, $\text{Zip}(K, \mathbb{A}, w_0)$ refers to the set of all action-labeled traces that are defined by paths in $\text{Paths}(K, \mathbb{A}, w_0)$.

We are now ready to formally define the central verification question solved in this paper, namely, the model-checking problem for specifications given as hypernode automata over models given as pointed Kripke structures with action labelling. The conversion of concurrent programs, such as those from Section II, into a pointed Kripke structure with action labeling is straightforward; its formalization is omitted here for space reasons.

Model-checking problem for hypernode automata

Let (K, w_0) be a pointed Kripke structure with set of variables X over a finite domain Σ, and let $\mathbb{A}$ be an action labeling for K over a set A of actions. Let $\mathcal{H}$ be a hypernode automaton over the same set X of variables, domain Σ and set A of actions. Is the set of action-labeled traces generated by $(K, \mathbb{A}, w_0)$ accepted by $\mathcal{H}$; that is, $\mathrm{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$?

4.2 Model Checking Hypernodes

We begin by formulating and solving the model-checking problem for hypernode logic (rather than automata) over Kripke structures. This algorithm constitutes the key subroutine for model-checking hypernode automata. To interpret formulas of hypernode logic over a Kripke structure, we equip the Kripke structure with two set of worlds: the *entry worlds*, where trace segments begin, and the *exit worlds*, where trace segments end. Formally, an *open Kripke structure* consists of a Kripke structure $K = (W, \Sigma^X, \Delta, V)$, and a pair $\mathbb{W} = (W_{\text{in}}, W_{\text{out}})$ consisting of a set $W_{\text{in}} \subseteq W$ of entry worlds, and a set $W_{\text{out}} \subseteq W$ of exit worlds.

A *path* of the open Kripke structure $(K, \mathbb{W})$ is path $w_0 \dots w_n$ in K that starts in a entry world, $w_0 \in W_{\text{in}}$ and ends in an exit world, $w_n \in W_{\text{out}}$. The set of unzipped trace segments generated by the open Kripke structure $(K, \mathbb{W})$ is $\mathrm{Unzip}(K)(x) = \{V(w_0, x) \dots V(w_n, x) \mid w_0 \dots w_n \in \mathrm{Paths}(K, \mathbb{W})\}$ for all variables $x \in X$.

Model-checking problem for hypernode logic

Let $(K, \mathbb{W})$ be an open Kripke structure, and φ a formula of hypernode logic over the same set of variables X and finite domain Σ. Is the set of unzipped trace segments generated by $(K, \mathbb{W})$ a model for φ; that is, $\mathrm{Unzip}(K, \mathbb{W}) \models \varphi$?

Stutter-free automata

From Proposition 2, it follows that it suffices to consider the stutter reduction of $\mathrm{Unzip}(K, \mathbb{W})$ to solve the model-checking problem for hypernode logic. We introduce *stutter-free automata* as a formalism for specifying sets of stutter-free unzipped trace segments. We use stutter-free automata boolean operators to define a filtration that, when applied to a hypernode formula φ and a stutter-free automaton over the variables in φ, returns an automaton with non-empty language iff the language of the input automaton is a model of φ. Finally, we construct, from a given open Kripke structure $(K, \mathbb{W})$, a stutter-free automaton that accepts an unzipped trace segment if the segment is the stutter reduction of a trace segment generated by $(K, \mathbb{W})$. We include a graphical overview of the algorithm in the extended version in [4].

Stutter-free automata are a restricted form of nondeterministic finite automata (NFA) that read unzipped trace segments and guarantees that, for each state, there are no repeated variable assignments on their incoming and outgoing transitions. We denote by Σ^X all assignments of variables in X to values in Σ or the termination symbol #. Formally, for $X = \{x_0, \dots, x_m\}$, let $\Sigma^X = \{x_0 : \sigma_0, \dots, x_m : \sigma_m \mid \forall 0 \leq i \leq m \ \sigma_i \in \Sigma \cup \{\#\}\} \setminus \{x_0 : \#, \dots, x_m : \#\}$.

▶ **Definition 5.** *Let X be a finite set of variables over Σ. A nondeterministic* stutter-free automaton *(NSFA) is a tuple $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ with a finite set Q of states, a set $\hat{Q} \subseteq Q$ of initial states, a set $F \subseteq Q$ of final states, and a transition relation $\delta : Q \times \Sigma^X \to 2^Q$ that satisfies the following for all states $q \in Q$ and variables $x \in X$: (i)* stutter-freedom *requiring $In(q, x) \cap Out(q, x) \subseteq \{\#\}$, and (ii)* termination *requiring that if $\# \in In(q, x)$,* *then $Out(q,x) = \{\#\}$, where $In(q,x)$ is the set of all x-valuations incoming to state q and $Out(q,x)$ is the set of all x-valuations outgoing from state q; formally, $In(q,x) = \{v(x) \mid q \in \delta(q',v)$ for some $q' \in Q\}$ and $Out(q,x) = \{v(x) \mid \delta(q,v) \neq \emptyset\}$.*

A *run* of the stutter-free automaton $\mathcal{A}$ is a finite sequence $q_0 v_0 q_1 v_1 \dots v_{n-1} q_n$ of alternating states and variable assignments which starts with an initial state, $q_0 \in \hat{Q}$, and satisfies the transition function, $q_{i+1} \in \delta(q_i, v_i)$ for all $i < n$. The run is *accepting* if it ends in a final state, $q_n \in F$. An unzipped trace segment τ over a set of variables X with domain Σ is *accepted* by the stutter-free automaton $\mathcal{A}$ iff there exists an accepting run $q_0 v_0 \dots v_{n-1} q_n$ such that $\tau(x) = v_0(x) \dots v_{n-1}(x)$, for all $x \in X$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all accepted unzipped trace segments accepted by $\mathcal{A}$. We sometimes refer to the language of a stutter-free automaton without the termination symbol: $\mathcal{L}(\mathcal{A})|_\# = \{\tau|_\# : X \to \Sigma^* \mid \tau \in \mathcal{L}(\mathcal{A})\}$, where $\tau|_\#$ removes all occurrences of $\#$ in a trace segment τ. Note that since $\mathcal{A}$ is stutter-free, $\mathcal{L}(\mathcal{A})|_\# = \lfloor \mathcal{L}(\mathcal{A})|_\# \rfloor$, where $\lfloor \cdot \rfloor$ is the stutter reduction of unzipped trace segments.

The union, intersection, and determinization for NSFA are defined as usual for NFA; we omit the formal definitions for reasons of space. The complementation of a stutter-free automaton follows the same approach as for NFA: we first determinize the automaton, then complete it, and lastly swap the final and nonfinal states. The only operation that requires special attention for NSFA is *completion*, as we need to be careful to statisfy the condition of stutter-freedom.

A stutter-free automaton $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ over Σ^X is *complete* iff $In(q) \cup Out(q)$ is a maximal subset of Σ^X according to the conditions in Definition 5, where $In(q) = \{v(x) \mid x \in X$ and $v(x) \in In(q,x)\}$ and $Out(q) = \{v(x) \mid x \in X$ and $v(x) \in Out(q,x)\}$. The *universal* stutter-free automaton $\mathcal{U}_{\Sigma^X}$ over Σ^X, defined next, is a deterministic and complete automaton with language $\mathcal{L}(\mathcal{U}_{\Sigma^X})|_\# = \lfloor (\Sigma^*)^X \rfloor$, i.e., it contains all stutter-free unzipped traces over Σ^X. We use the universal stutter-free automaton as a "sink" area when completing other automata.

▶ **Definition 6.** *Let $X = \{x_0, \dots, x_m\}$ be a set of variables over the finite domain Σ. The* universal stutter-free automaton *over Σ^X is $\mathcal{U}_{\Sigma^X} = (Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$, where $Q_\mathcal{U} = \Sigma^X$ and*

$$\delta_\mathcal{U}(\{x_i : \sigma_i\}_{i\in[0,m]}, \{x_i : \sigma'_i\}_{i\in[0,m]}) = \begin{cases} \{x_i : \sigma'_i\}_{i\in[0,m]} & \textit{if } \forall 0 \le i \le m,\ \textit{if } \sigma_i = \# \textit{ then } \sigma'_i = \# \textit{ else } \sigma_i \neq \sigma'_i; \\ \emptyset & \textit{otherwise.} \end{cases}$$

We use the states and transitions of the universal automaton to complete other stutter-free automata. The details are in the extended version in [4]. The complement of a deterministic and complete stutter-free automaton $\mathcal{A} = (Q, \hat{Q}, F, \delta)$ over Σ^X is $\overline{\mathcal{A}} = (Q, \hat{Q}, Q \setminus F, \delta)$, with the final and nonfinal states interchanged.

▶ **Proposition 7.** *Let $\mathcal{A}$ be a deterministic and complete stutter-free automaton over Σ^X. Then, $\overline{\mathcal{A}}$ is a stutter-free automaton and $\mathcal{L}(\overline{\mathcal{A}}) = \lfloor (\Sigma^*)^X \rfloor \setminus \mathcal{L}(\mathcal{A})$.*

From formulas of hypernode logic to stutter-free automata

Having prepared the ground by defining stutter-free automata, which are closed under union, intersection, and complement, we now turn to the model-checking problem for hypernode logic. Given a stutter-free automaton and a hypernode formula, we define an inductive filtration (i.e., in each step we get produce a sub-automaton) over the hypernode formula structure to apply to the automaton we want to model-check. The input automaton is a model of the input formula if the language of the automaton returned by the filtration is non-empty. The inductive filtration for boolean operators translates naturally to automata operators. For atomic hypernode formulas (i.e., the predicate $\precsim$), we define a stutter-free automaton that captures the meaning of $\precsim$.

▶ **Definition 8.** *Let* $\mathcal{U}_X = (Q_\mathcal{U}, Q_\mathcal{U}, Q_\mathcal{U}, \delta_\mathcal{U})$ *be the universal stutter-free automaton over* Σ^X*, and let* $x, y \in X$*. The* stutter-free automaton for the atomic formula $x \precsim y$ of hypernode logic *is the stutter-free automaton* $\mathcal{A}_{x \precsim y} = (Q, Q, Q, \delta)$ *over the same variables and domain, where* $Q = \{v \in Q_\mathcal{U} \mid v(x) = v(y)$ *or* $v(x) = \#\}$*, and* $\delta(q, v) = \delta_\mathcal{U}(q, v)$ *for all* $q \in Q$ *and* $v \in \Sigma^X$*.*

To cope with trace variables in hypernode formulas, we extend the set of variables X with a reference to trace variables in $\mathcal{V}$, by $X_\mathcal{V} = \{x_\pi \mid x \in X \text{ and } \pi \in \mathcal{V}\}$. From an unzipped trace segment τ over the set of variables $X_\mathcal{V}$ we derive the trace assignment $\Pi_\tau(\pi, x) = \tau(x_\pi)$ for all $x \in X$ and $\pi \in \mathcal{V}$. We prove now that all words accepted by the stutter-free automaton for $x(\pi) \precsim y(\pi')$ define assignments that satisfy that hypernode atomic formula.

▶ **Lemma 9.** *An unzipped trace segment* τ *over* $(\Sigma^{X_\mathcal{V}})^*$ *is accepted by* $\mathcal{A}_{x_\pi \precsim y_{\pi'}}$ *over the same variables and domain,* $\tau \in \mathcal{L}(\mathcal{A}_{x_\pi \precsim y_{\pi'}})|_\#$*, iff* $\Pi_\tau \models x(\pi) \precsim y(\pi')$*.*

The inductive filtration defined next is the main element of the model-checking algorithm for formulas of hypernode logic.

▶ **Definition 10.** *Let* $\mathcal{A}$ *be a stutter-free automaton, and* φ *a formula of hypernode logic. We define the positive and negative filtration of* $\mathcal{A}$ *by* φ*, denoted* $\varphi^+[\mathcal{A}]$ *and* $\varphi^-[\mathcal{A}]$*,respectively, inductively over the structure of* φ *as follows:*

$$\begin{aligned}
(x(\pi) \precsim y(\pi'))^+[\mathcal{A}] &= \mathcal{A} \cap \mathcal{A}_{x_\pi \precsim y_{\pi'}} & (x(\pi) \precsim y(\pi'))^-[\mathcal{A}] &= \mathcal{A} \cap \overline{\mathcal{A}_{x_\pi \precsim y_{\pi'}}} \\
(\varphi_1 \wedge \varphi_2)^+[\mathcal{A}] &= \varphi_1^+[\mathcal{A}] \cap \varphi_2^+[\mathcal{A}] & (\varphi_1 \wedge \varphi_2)^-[\mathcal{A}] &= \varphi_1^-[\mathcal{A}] \cup \varphi_2^-[\mathcal{A}] \\
(\neg\varphi)^+[\mathcal{A}] &= \varphi^-[\mathcal{A}] & (\neg\varphi)^-[\mathcal{A}] &= \varphi^+[\mathcal{A}] \\
(\exists\pi\varphi)^+[\mathcal{A}] &= \varphi^+[\mathcal{A}] & (\exists\pi\varphi)^-[\mathcal{A}] &= \mathcal{A} \setminus \varphi^+[\mathcal{A}].
\end{aligned}$$

We reduce the problem of model checking a stutter-free automaton $\mathcal{A}$ over a formula φ with n trace variables to filtering the n-self-composition of $\mathcal{A}$ by φ. The stutter-free automaton $\mathcal{A}^n$ is the result of composing n copies of $\mathcal{A}$ under a standard synchronous product construction, where for each copy $\mathcal{A}_i$, with $i \leq n$, all program variables $x \in X$ are renamed to x_{π_i}. Note that, the assignment derived by an unzipped trace segment τ accepted by $\mathcal{A}^n$ defines a trace assignment from $\{\pi_1, \ldots, \pi_n\}$ to traces accepted by $\mathcal{A}$.

▶ **Theorem 11.** *Let* $\mathcal{A}$ *be a stutter-free automaton, and* φ *a formula of hypernode logic with* n *trace variables. Then,* $\mathcal{L}(\varphi^+[\mathcal{A}^n]) \neq \emptyset$ *iff* $\mathcal{L}(\mathcal{A}) \models \varphi$*, and* $\mathcal{L}(\varphi^-[\mathcal{A}^n]) \neq \emptyset$ *iff* $\mathcal{L}(\mathcal{A}) \not\models \varphi$*.*

Model checking hypernode logic over Kripke structures

We are only missing to translate an open Kripke structure $(K, \mathbb{W})$ to a stutter-free automaton $\mathcal{A}_{K,\mathbb{W}}$ defining the same unzipped trace segments; i.e., $\mathcal{L}(\mathcal{A}_{K,\mathbb{W}})|_\# = \lfloor \text{Unzip}(K, \mathbb{W}) \rfloor$. We present the details in the appendix, but, in a nutshell, we represent the progression of each variable valuation along the Kripke structure independently (to allow skipping stuttering states) in the derived stutter-free automaton. Having this translation, we can apply the filtration from Definition 10 to the stutter-free automaton derived by an open Kripke structure to solve the model-checking problem for hypernode logic.

▶ **Theorem 12.** *Let $(K, \mathbb{W})$ be an open Kripke structure, and φ a formula of hypernode logic over the same set of variables. Let n be the number of trace variables in φ. Then,* $\mathrm{Unzip}(K, \mathbb{W}) \models \varphi$ *iff* $\mathcal{L}(\varphi^{+}[\mathcal{A}^{n}_{K,\mathbb{W}}]) \neq \emptyset$.

The proof follows from Proposition 7, Theorem 11, and $\mathcal{L}(\mathcal{A})|_{\#} = \lfloor \mathcal{L}(\mathcal{A})|_{\#} \rfloor$. This gives us our main result.

▶ **Theorem 13.** *Model checking of hypernode logic over open Kripke structures is decidable.*

Using our algorithm, the running time of model checking a formula of hypernode logic over an open Kripke structure depends doubly exponentially on the number of variables, singly exponentially on the number of worlds of the Kripke structure, and singly exponentially on the length of the formula.

▶ **Corollary 14.** *The time complexity of model checking a formula φ of hypernode logic with n trace variables and m variables, over an open Kripke structure with k worlds, is $\mathcal{O}(2^{n \cdot k^{m}})$.*

Proof. The encoding of the open Kripke Structure by a stutter-free automaton has $\mathcal{O}(k^{m})$ states. The determinized stutter-free automaton has $\mathcal{O}(2^{k^{m}})$ states. After completing the deterministic stutter-free automaton there are 2^{m} states. We observe that the size of the domain Σ only affects the step of completing a stutter-free automaton, which adds $|\Sigma|^{|X|}$ states. This addition is dominated by the more expensive step of determinizing the automaton. Finally, the n-self-composition of the resulting automaton has $\mathcal{O}(2^{n \cdot k^{m}})$ states. ◀

4.3 Model Checking Hypernode Automata

We defined the run of a hypernode automaton for a given action sequence p, with each run inducing a slicing of a set of action-labeled traces consistent with p. To model-check a hypernode automaton $\mathcal{H}$ against a pointed Kripke structure $(K, \mathbb{A}, w_0)$ with an action labeling, we build a finite automaton, called $\mathrm{Slice}(K, \mathbb{A}, w_0)$, which encodes all slicings of action-labeled traces generated by $(K, \mathbb{A}, w_0)$. We then reduce the model-checking problem to checking whether the language defined by the composition of $\mathrm{Slice}(K, \mathbb{A}, w_0)$ with the specification automaton $\mathcal{H}$, called $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$, is non-empty. We include an overview of this process in the extended version in [4].

We start by defining the *slicing* of a given Kripke structure $K = (W, \Sigma^{X}, \Delta, V)$ for a given action labeling $\mathbb{A}$. The building blocks of the slicing are Kripke substructures. A Kripke structure $K' = (W', \Sigma^{X}, \Delta', V')$ is a *substructure* of K, denoted $K' \leq K$, iff $W' \subseteq W$, and for all worlds $w \in W'$ we have $\Delta'(w) \subseteq \Delta(w)$ and $V'(w) = V(w)$. The *substructure induced by a transition relation* $\Delta' \subseteq \Delta$ is $K[\Delta'] = (W', X, \Delta', V(W'))$, where $W' = \{w, w' \mid (w, w') \in \Delta'\}$. The transition relation defined by *all transitions in a path* of the action-labeled Kripke structure $(K, \mathbb{A})$ from an entry world in $W_{\mathrm{in}} \subseteq W$ to the first step labeled with $a \in A$ is:

$$(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a = \{(w_j, w_{j+1}) \mid w_0 \varepsilon \ldots w_{n-1} \varepsilon\, w_n a \in \mathrm{Paths}(K, \mathbb{A}), w_0 \in W_{\mathrm{in}} \text{ for all } j < n\}.$$

The *open substructure induced by* $(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a$, written $\mathbb{W}[(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a]$, is the open Kripke structure where the Kripke structure is $K[(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a]$, the set W_{in} are the entry worlds, and the set $\{w \mid w \in W \text{ and } \mathbb{A}(w, a) \neq \emptyset\}$ are the exit worlds, containing all possible exit points for action a.

We define the finite automaton $\mathrm{Slice}(K, \mathbb{A}, w_0)$ and prove, in Lemma 16 below, that every finite action sequence p defines a unique path in this automaton, and the slices of this path contain the same trace segments that are obtained when the action sequence p is applied directly to the original pointed, action-labeled Kripke structure. The states of the automaton

$\mathrm{Slice}(K, \mathbb{A}, w_0)$ are all open substructures induced by paths from any choice of entry worlds to an action $a \in \mathbb{A}$. Note that there are only finitely many such states. The transition relation of $\mathrm{Slice}(K, \mathbb{A}, w_0)$ connects, for all actions a, open substructures with exit a and open substructures with matching entry worlds.

▶ **Definition 15.** *Let (K, w_0) be a pointed Kripke structure with worlds W, and let $\mathbb{A}$ be an action labeling for K with actions A. The slicing* $\mathrm{Slice}(K, \mathbb{A}, w_0) = (Q, \hat{Q}, \delta)$ *is a finite automaton with states $Q = \{\mathbb{W}[(K, \mathbb{A}, W_{\mathrm{in}}) \downarrow a] \mid a \in A$ and $W_{\mathrm{in}} \subseteq W\}$; initial states $\hat{Q} = \{\mathbb{W} \in Q \mid \mathrm{entry}(\mathbb{W}) = \{w_0\}\}$; transition function $\delta : Q \times A \to Q$, where $\delta(\mathbb{W}, a) = \mathbb{W}'$ iff $\mathbb{W}$ exits with action a, that is, for all $w \in \mathrm{exit}(\mathbb{W})$ there exists $w' \in \mathbb{W}'$ such that $a \in \mathbb{A}(w, w')$, and the entry worlds of $\mathbb{W}'$ define a maximal subset of the worlds accessible with action a from the exit worlds in $\mathbb{W}$, that is, for all $\mathbb{W}'' \in Q$ that are not $\mathbb{W}'$, if $\mathrm{entry}(\mathbb{W}'') \subseteq \{w \mid a \in \mathbb{A}(w', w)$ for some $w' \in \mathrm{exit}(\mathbb{W})\}$, then $\mathrm{entry}(\mathbb{W}') \not\subseteq \mathrm{entry}(\mathbb{W}'')$. Here, $\mathrm{entry}(\mathbb{W})$ and $\mathrm{exit}(\mathbb{W})$ refer to the sets of entry and exit worlds of the open Kripke structure $\mathbb{W}$, respectively.*

We remark that the transition function $\delta : Q \times A \to Q$ is well-defined, because there is a unique maximal subset for the next entry worlds, given an action a. For every two open Kripke substructures, $\mathbb{W}_1$ and $\mathbb{W}_2$, their union defines $\mathbb{W}[(K, \mathbb{A}, \mathrm{entry}(\mathbb{W}_1) \cup \mathrm{entry}(\mathbb{W}_2)) \downarrow a]$, which is again a state of the slicing.

▶ **Lemma 16.** *Let (K, w_0) be a pointed Kripke structure, and $\mathbb{A}$ an action labeling for K with actions A. For every finite action sequence $p = a_0 \ldots a_n$ in A^*, if $\mathrm{Zip}(K, \mathbb{A}, w_0)[p] \neq \emptyset$, then p defines a unique run $\mathbb{W}_0 a_0 \cdots \mathbb{W}_n a_n$ of $\mathrm{Slice}(K, \mathbb{A}, w_0)$ such that for all $0 \leq i \leq n$, $\mathrm{Paths}(\mathbb{W}_i) = \mathrm{Paths}(K, \mathbb{A}, w_0)(a_0 \ldots a_{i-1}, a_i)$.*

In a final step, we define a synchronous composition of the slicing automaton defined above and the given hypernode automaton $\mathcal{H}$. The states of this composition are pairs consisting of open Kripke substructures (stemming from the given pointed, action-labeled Kripke structure) and formulas of hypernode logic (stemming from the hypernode labels of $\mathcal{H}$). We mark as final states all pairs where the open Kripke substructure is not a model of the hypernode formula.

▶ **Definition 17.** *Let $\mathcal{H} = (Q_h, \hat{q}, \gamma, \delta_h)$ be a hypernode automaton. The* intersection of $\mathcal{H}$ with the slicing of a pointed, action-labeled Kripke structure *$(K, \mathbb{A}, w_0)$*, $\mathrm{Slice}(K, \mathbb{A}, w_0) = (Q_s, \hat{Q}_s, \delta_s)$, *is the finite automaton* $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0) = (Q, \hat{Q}, F, A, \delta)$ *with set of states $Q = \{(\mathbb{W}, q) \mid \mathbb{W} \in Q_s, q \in Q_h$ and $\mathbb{W} \models \gamma(q)\} \cup \{(\mathbb{W}, \overline{q}) \mid \mathbb{W} \in Q_s, q \in Q_h$ and $\mathbb{W} \not\models \gamma(q)\}$; initial states $\hat{Q} = \{(\mathbb{W}, \hat{q}) \in Q \mid \mathbb{W} \in \hat{Q}_s\} \cup \{(\mathbb{W}, \bar{\hat{q}}) \in Q \mid \mathbb{W} \in \hat{Q}_s\}$; final state $F = \{(\mathbb{W}, \overline{q}) \mid (\mathbb{W}, \overline{q}) \in Q\}$; transition function $\delta : Q \times A \to Q$, where for all $(\mathbb{W}, q) \in Q$, we have $\delta((\mathbb{W}, q), a) = \{(\mathbb{W}', q') \in Q \mid \delta_h(q) = (q', a)$ and $\mathbb{W}' \in \delta_s(\mathbb{W}, a)\}$.*

The finite automaton $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ reads sequences of actions. The notion of run is defined as usual, and a run is accepting if it ends in a final state. The language of the automaton is empty iff it has no accepting run.

▶ **Theorem 18.** *Let (K, w_0) be a pointed Kripke structure with action labeling $\mathbb{A}$. Let $\mathcal{H}$ be a hypernode automaton over the same set of propositions and actions as $(K, \mathbb{A})$. Then,* $\mathrm{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$ *iff the language of the finite automaton* $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ *is empty.*

The following theorem puts all results from this section together.

▶ **Theorem 19.** *Model checking of hypernode automata over pointed Kripke structures with action labelings is decidable.*

Proof. We have seen that the model checking of hypernode logic over open Kripke structures is decidable (Theorem 13). Evaluating $\mathrm{Join}(\mathcal{H}, K, \mathbb{A}, w_0)$ is also decidable. The main challenge is the slicing of $\mathbb{A}(K, w_0)$. Note that there is a finite number of states that can be in $\mathbb{A}(K, w_0)$, as they are all substructures of the Kripke structure K. ◀

The hardest part of model checking a hypernode automaton over an action-labeled Kripke structure is checking the formulas of all hypernodes. Therefore, also the running time for model checking hypernode automata is dominated, as with hypernode logic, by a doubly exponential dependency on the number of program variables. Furthermore, our model-checking algorithm depends singly exponentially on both the size of the Kripke structure and the size of the hypernode automaton.

▶ **Corollary 20.** *Let A be a set of actions and X a set of m program variables. Let (K, w_0) be a pointed Kripke structure over X, and $\mathbb{A}$ an action labeling for K over A. Let $\mathcal{H}$ be a hypernode automaton over X and A. The time complexity of checking whether $\mathrm{Zip}(K, \mathbb{A}, w_0) \in \mathcal{L}(\mathcal{H})$ is $\mathcal{O}(|\mathcal{H}| \cdot 2^{|A|+n\cdot|K|^m})$, where n is the largest number of trace quantifiers that occurs in any hypernode formula in $\mathcal{H}$.*

5 Related Work

The first logic studied to express asynchronous hyperproperties was an extension of μ-calculus with explicit quantification over traces, called Hμ [12]. The trace-quantifier free formulas of Hμ are expressively equivalent to the parity multi-tape Alternating Asynchronous Word Automata (AAWA) introduced in [12]. Both formalisms have highly undecidable model-checking problems. The undecidability stems from comparing positions in different traces that are arbitrarily far apart, over an unbounded number of traces. When one of the two dimensions (the distance between positions, or the number of traces) is given an explicit finite bound, model checking becomes decidable [12]. In comparison, we achieve decidability by an entirely different means: we decouple the progress of different program variables (asynchronicity), while allowing resynchronization through automaton-level transitions.

In Hμ formulas, trace quantifiers always precede time operators, while hypernode automata allow a restricted form of quantifier alternation between time operators and trace quantifiers. In particular, the automaton-level transitions correspond to outermost time operators, which precede the trace quantifiers of hypernode logic formulas, whose stutter-reduced prefixing relations correspond to innermost time quantifiers. We conjecture that Hμ and hypernode automata have incomparable expressive powers. Consider, for example, the hypernode automaton shown in Figure 2, which specifies that the asynchronous progress of a propositional variable p is fully described by a finite trace π within each slice induced by a repeated action a. Each new slice can have a different trace π witnessing the asynchronous progress of p. The length of the traces in each slice is unbounded, and as we do not know how many times a repeats, the number of slices is also unbounded. Hence we do not know how many outermost existential trace quantifiers would be needed in order to guarantee a different trace witness for each slice. Therefore we conjecture that the hyperproperty that is specified by the hypernode automaton of Figure 2 cannot be expressed in Hμ.

Also various extensions of HyperLTL were explored recently in order to support asynchronous hyperproperties. *Stuttering* HyperLTL (HyperLTL$_S$) and *context* HyperLTL (HyperLTL$_C$), both introduced in [8], extend HyperLTL with new operators. *Asynchronous* HyperLTL (A-HyperLTL) [5] extends HyperLTL with quantification over *trajectories.* A trajectory specifies the traces that progress in each evaluation step. While the model-checking

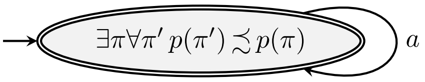


Figure 2 Hypernode automaton specifying that within each slice of a trace set induced by the repeated action a, there exists a trace that describes the asynchronous progress of the propositional variable p within the current slice.

problems for all of these extensions of HyperLTL are undecidable, the authors identify syntactic fragments that support certain asynchronous hyperproperties. These decidable fragments adopt restrictions akin to the decidable parts of Hμ. All of HyperLTL$_S$, HyperLTL$_C$, and A-HyperLTL are subsumed by Hμ [9]. As we argued that the hypernode automaton from Figure 2 cannot be expressed in Hμ, it would neither be expressible in any of the three proposed asynchronous extensions of HyperLTL.

Krebs et al. [14] propose to reinterpret LTL under a so-called *team* semantics. Team semantics works with sets of variable assignments, and the authors introduce both synchronous and asynchronous varieties. They prove that under asynchronous team semantics, LTL is as expressive as universal HyperLTL (where all trace quantifiers are universal quantifiers). As hypernode logic allows existential quantification over traces, again, our approach is orthogonal and expressively incomparable.

In [6] the authors introduce HyperATL*, which extends alternating-time temporal logic [1] with strategy quantifiers that bind strategies to trace variables, and an explicit construct to resolve games in parallel. HyperATL* enables the specification of strategic hyperproperties. This work is orthogonal to ours as we are interested in *linear-time* asynchronous hyperproperties, rather than strategic hyperproperties.

Although many logic-based specification languages have been proposed to express asynchronous hyperproperties, there is a lack of automaton-based approaches to specify such properties. Note that AAWA [12] do not support explicit quantification over trace variables. The finite-word *hyperautomata* of [7] constitute a step in this direction by prefixing finite automata with explicit quantification over traces, but they are limited to *synchronous* hyperproperties.

6 Conclusion

We presented a new formalism for specifying hyperproperties of concurrent systems. Our formalism mixes synchronization between different execution traces, expressed as action-labeled transitions of a specification automaton, with asynchronous comparisons between corresponding segments of different traces, expressed as hypernode logic formulas that label the states of the specification automaton. In this way, the specification language of hypernode automata can alternate asynchronous requirements on trace segments of possibly different lengths with synchronization points. Unlike previous formalisms for specifying asynchronous hyperproperties, hypernode automata fully support automatic verification. Our model-checking algorithm for hypernode automata is based on an entirely novel technique that introduces stutter-free automata and operations on these automata, thus providing a nice example for the power of automata-theoretic methods in verification.

Besides having a decidable verification problem, hypernode automata represent a genuinely new and useful specification language. We demonstrated this by specifying several published variations of observational determinism using hypernode logic, by specifying information declassification using hypernode automata, and by presenting a formula to support our conjecture that the formalism of hypernode automata is expressively incomparable to various hyperlogics that have been proposed recently for specifying asynchronous hyperproperties.

The boundary between asynchronicity and synchronicity of trace comparisons can be fine-tuned by introducing variables with compound types, such as boolean arrays, which can be used, for example, to couple the variables of each thread of a multi-threaded program. The ramifications of such alphabet variations on hypernode logic and hypernode automata are to be explored in future work. There is no shortage of additional topics that follow immediately from the present work but, even if straightforward, require further investigations, including the study of hypernode automata with partial and nondeterministic transition relations, and of hypernode automata with infinitary acceptance conditions (such as hypernode Büchi automata), as well as the extension of formal expressiveness studies for hyperproperty specifications in order to include hypernode logic and automata, and the presentation of algorithms for solving classical decision problems for hypernode logic and automata other than model checking (such as satisfiability and emptiness). Also the applicability of stutter-free automata in other asynchronous verification contexts (not necessarily concerning hyperproperties) is an interesting question.

References

1 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002. doi:10.1145/585265.585270.

2 Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy*, pages 207–221, 2007. doi:10.1109/SP.2007.22.

3 Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Flavors of sequential information flow. In Bernd Finkbeiner and Thomas Wies, editors, *23rd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 13182 of *LNCS*, pages 1–19. Springer International Publishing, 2022. doi:10.1007/978-3-030-94583-1_1.

4 Ezio Bartocci, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Hypernode automata, 2023. arXiv:2305.02836.

5 Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, pages 694–717. Springer International Publishing, 2021. doi:10.1007/978-3-030-81685-8_33.

6 Raven Beutner and Bernd Finkbeiner. A Temporal Logic for Strategic Hyperproperties. In *32nd International Conference on Concurrency Theory (CONCUR)*, volume 203 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2021.24.

7 Borzoo Bonakdarpour and Sarai Sheinvald. Finite-word hyperlanguages. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications (LATA)*. Springer International Publishing, 2021. doi:10.1007/978-3-030-68195-1.

8 Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of HyperLTL. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. Association for Computing Machinery, 2021. doi:10.1109/LICS52264.2021.9470583.

9 Laura Bozzelli, Adriano Peron, and César Sánchez. Expressiveness and Decidability of Temporal Logics for Asynchronous Hyperproperties. In Bartek Klin, Sławomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2022.27.

10 Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proc. of POST 2014: the Third International Conference on Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8.

11 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

12 Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL), 2021. doi:10.1145/3434319.

13 M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 13 pp.–3, 2006. doi:10.1109/CSFW.2006.6.

14 Andreas Krebs, Arne Meier, Jonni Virtema, and Martin Zimmermann. Team Semantics for the Specification and Verification of Hyperproperties. In Igor Potapov, Paul Spirakis, and James Worrell, editors, *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018)*, volume 117 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.MFCS.2018.10.

15 Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

16 Tachio Terauchi. A type system for observational determinism. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300, 2008. doi:10.1109/CSF.2008.9.

17 S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, pages 29–43, 2003. doi:10.1109/CSFW.2003.1212703.

Quantitative Verification with Neural Networks

Alessandro Abate
University of Oxford, UK

Alec Edwards
University of Oxford, UK

Mirco Giacobbe
University of Birmingham, UK

Hashan Punchihewa
University of Oxford, UK

Diptarko Roy
University of Oxford, UK

Abstract

We present a data-driven approach to the quantitative verification of probabilistic programs and stochastic dynamical models. Our approach leverages neural networks to compute tight and sound bounds for the probability that a stochastic process hits a target condition within finite time. This problem subsumes a variety of quantitative verification questions, from the reachability and safety analysis of discrete-time stochastic dynamical models, to the study of assertion-violation and termination analysis of probabilistic programs. We rely on neural networks to represent supermartingale certificates that yield such probability bounds, which we compute using a counterexample-guided inductive synthesis loop: we train the neural certificate while tightening the probability bound over samples of the state space using stochastic optimisation, and then we formally check the certificate's validity over every possible state using satisfiability modulo theories; if we receive a counterexample, we add it to our set of samples and repeat the loop until validity is confirmed. We demonstrate on a diverse set of benchmarks that, thanks to the expressive power of neural networks, our method yields smaller or comparable probability bounds than existing symbolic methods in all cases, and that our approach succeeds on models that are entirely beyond the reach of such alternative techniques.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Probabilistic computation; Computing methodologies → Machine learning; Computing methodologies → Neural networks

Keywords and phrases Data-driven Verification, Quantitative Verification, Probabilistic Programs, Stochastic Dynamical Models, Counterexample-guided Inductive Synthesis, Neural Networks

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.22

Related Version *FullVersion*: `https://arxiv.org/abs/2301.06136` [2]

Funding *Alessandro Abate*: HICLASS project from Innovate UK
Alec Edwards: EPSRC Centre for Doctoral Training in Autonomous Intelligent Machines and Systems (EP/S024050/1)
Hashan Punchihewa: DeepMind Computer Science Scholarship
Diptarko Roy: EPSRC Doctoral Training Partnership, and Department of Computer Science Scholarship, University of Oxford

1 Introduction

Probabilistic programs extend imperative programs with the ability to sample from probability distributions [20, 27, 31, 37], which provides an expressive language to describe randomized algorithms, cryptographic protocols, and Bayesian inference schemes. Discrete-time stochastic dynamical models, characterised by stochastic difference equations, are a natural framework

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 22; pp. 22:1–22:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to describe auto-regressive time series, as well as sequential decision and planning problems in unknown environments. A fundamental quantitative verification problem for probabilistic programs and stochastic dynamical models is the quantitative reachability question, which amounts to finding the probability with which the system reaches a given target condition within a finite number of steps. Reachability is at the core of a variety of other important quantitative verification questions as, by selecting appropriate target conditions on the state space, we can express the probability that a probabilistic program terminates or that it violates an assertion, as well as the probability that a stochastic dynamical model satisfies an invariant or remains within a set of safe configurations.

Quantitative reachability verification has been studied extensively using theories and algorithms built upon symbolic reasoning techniques, such as quantitative calculi [36,38], probabilistic model checking algorithms [22,33], discrete abstractions of stochastic dynamical models [4,44], and the synthesis of supermartingale-like certificates [13,15,16,19]. Among the latter class, a method to provide a sound upper bound for the reachability probability of a system is to synthesise a supermartingale function that maps every reachable state to a non-negative real, whose value is never smaller than 1 inside the target condition, and such that it never increases in expectation as the system evolves outside of the target. This is referred to as a *non-negative repulsing supermartingale* or *stochastic invariant indicator* in the literature [17,19,43], and its output over a given state provides an upper bound for the probability that the system reaches the target condition from that state. Symbolic methods for the synthesis of such certificates assume that the supermartingale function, as well as its post-expectation, which depends on the model constraints and distributions, are both in linear or polynomial form. This poses syntactic restrictions to their applicability.

Data-driven and counterexample-guided inductive synthesis (CEGIS) procedures, combined with machine learning approaches that leverage neural networks to represent certificates, have shown great promise in mitigating the aforementioned limitation [1,3,14,26,34,35,40]. In particular, neural-based CEGIS decouples the task of guessing a certificate from that of checking its validity, delegating the guessing task to efficient machine learning algorithms that leverage the expressive power of neural networks, while confining symbolic reasoning to the checking part of the task, which is computationally easier to solve in isolation than the entire synthesis problem. CEGIS has been applied to the synthesis of *neural supermartingales* for the almost-sure termination of probabilistic programs [3], as well as their counterpart for stochastic dynamical models with applications to qualitative queries such as almost-sure safety and stability [34,35].

In this paper, we present theory, methods, and an extensive experimental evaluation to demonstrate the efficacy and flexibility of neural supermartingales to solve *quantitative verification* questions for probabilistic program and stochastic dynamical models, using machine learning combined with satisfiability modulo theories (SMT) technologies.

Theory. We adapt the theory of non-negative repulsing supermartingales to leverage neural networks as representations of supermartingale functions for quantitative verification. Unlike previous deductive methods which need deterministic invariants and impose restrictions on the models under study, our version entirely relies on neural architectures and SMT solving to guarantee soundness, without requiring such assumptions.

Methods. We present a CEGIS-based approach to train neural supermartingale functions that minimise an upper bound for the reachability probability over sample points from the state space, and check their validity over every possible state using SMT solving. We present a program-agnostic approach that relies on state samples in the training phase, and also a novel program-aware approach that embeds model information in the loss function to enhance the effectiveness of stochastic optimisation.

$$\begin{array}{lr}
v \in \text{Vars} & \text{(variables)} \\
N \in \mathbb{R} & \text{(numerals)} \\
E ::= v \mid N \mid \texttt{-}E \mid E + E \mid E - E \mid E * E \mid \ldots & \text{(arithmetic expressions)} \\
P ::= \texttt{Bernoulli(}\,E\,\texttt{)} \mid \texttt{Gaussian(}\,E\texttt{,}\ E\,\texttt{)} \mid \ldots & \text{(probability distributions)} \\
B ::= \texttt{true} \mid \texttt{!}B \mid B\,\texttt{\&\&}\,B \mid B\,\texttt{||}\,B \mid E\,\texttt{==}\,E \mid E\,\texttt{<}\,E \mid \ldots & \text{(Boolean expressions)} \\
C ::= \texttt{skip} & \text{(update commands)} \\
\qquad \mid v\,\texttt{=}\,E & \text{(deterministic assignment)} \\
\qquad \mid v\ \sim\ P & \text{(probabilistic assignment)} \\
\qquad \mid C\ ;\ C & \text{(sequential composition)} \\
\qquad \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi} & \text{(conditional composition)}
\end{array}$$

Figure 1 Grammar for update commands, Boolean and arithmetic expressions.

Experiments. We build a prototype implementation and compare the efficacy of our method with the state of the art in synthesis of linear supermartingales using symbolic reasoning [43]. We show that our program-aware approach computes tighter or comparable probability bounds than symbolic reasoning on existing benchmarks, while our program-agnostic approach matches it in over half of the instances. We additionally demonstrate that both our approaches can handle models beyond reach of purely symbolic methods.

2 Probabilistic Programs and Stochastic Dynamical Models

Probabilistic programs are computer programs whose execution is determined by random variables, and stochastic dynamical models describe discrete-time dynamical systems with probabilistic behaviour. The earlier enjoy the flexibility of imperative programming constructs and are used to describe randomised algorithms, and the latter are expressed as stochastic difference equations and are used to describe probabilistic systems that evolve over infinite time. The semantics of both can be described in terms of stochastic processes and, for this reason, verification questions for both can be solved with similar techniques.

The syntax of our modeling framework uses imperative constructs from probabilistic programs and defines executions over infinite time as dynamical systems. Specifically, we consider programs that operate over an ordered set of n real-valued variables, denoted by Vars, and update their values through the repeated execution of a command C whose grammar is described in Figure 1. Under this definition, a state of the system is an n-dimensional vector $s \in \mathbb{R}^n$ that assigns a value to each variable symbol. The update command C defines an update function $f\colon \mathbb{R}^n \times [0,1]^m \to \mathbb{R}^n$, where m is the number of syntactic probabilistic assignment statements occurring in C, which maps the current state and m random variables uniformly distributed in $[0,1]$ into the next state. Conceptually, within f, each random variable is mapped into its respective distribution by applying the appropriate inverse transformation. Altogether, our probabilistic program defines a stochastic process, whose behavior is determined by the following stochastic difference equation:

$$s_{t+1} = f(s_t, r_t), \quad r_t \sim \mathbb{U}^m, \tag{1}$$

where r_t is an m-dimensional random input sampled at time t from the uniform distribution $\mathbb{U}^m$ over the m-dimensional hypercube $[0,1]^m$. The initial state s_0 is either given as a deterministic assignment to constant values, or is non-deterministically chosen from a set of initial conditions S_0 characterized by a Boolean expression. This setting can be seen as an assertion about the initial conditions followed by the probabilistic program `while true do` C `od`. As we show in Section 3, this allows us to characterise verification questions such as termination, non-termination, invariance and assertion-violation for while loops with general guards, as well as reachability and safety verification questions for dynamical models with general stochastic disturbances.

The semantics of our model is defined as a stochastic process induced by the Markov chain over the probability space of infinite words of random samples. This is defined by the probability space triple $(\Omega, \mathcal{F}, \mathbb{P})$ [12, 28], where

- Ω is the set of infinite sequences $([0,1]^m)^\omega$ of m-dimensional tuples of values in $[0,1]$,
- $\mathcal{F}$ is the extension of the Borel σ-algebra over the unit interval $\mathcal{B}([0,1])$ to Ω,
- $\mathbb{P}$ is the extension of the Lebesgue measure on $[0,1]$ to Ω.

Every initialisation of the system on state $s \in \mathbb{R}^n$ induces a stochastic process $\{X_t^s(\omega)\}_{t\in\mathbb{N}}$ over the state space $\mathbb{R}^n$. Let $\omega = r_0 r_1 r_2 \dots$ be an infinite sequence of random samples in $[0,1]^m$, then the stochastic process is defined by the sequence of random variables

$$X_{t+1}^s(\omega) = f(X_t^s(\omega), r_t), \qquad X_0^s(\omega) = s. \tag{2}$$

This defines the natural filtration $\{\mathcal{F}_t\}_{t\in\mathbb{N}}$, which is the smallest filtration to which the stochastic process X_t^s is adapted. In other words [30], this can be seen as another Markov chain with state space $\mathbb{R}^n$ and transition kernel

$$T(s, S') = \mathrm{Leb}\left(\{r \in [0,1]^m \mid f(s,r) \in S'\}\right), \tag{3}$$

where S' is a Borel measurable subset of $\mathbb{R}^n$ and Leb refers to the Lebesgue measure of a measurable subset of $[0,1]^m$. In other words, kernel T denotes the probability to transition from state s into a set of states S'. The transition kernel also defines the *post-expectation operator* $\mathbb{X}$, also known as the next-time operator [43, Definition 2.16]. $\mathbb{X}$ can be applied to an arbitrary Borel-measurable function $h\colon \mathbb{R}^n \to \mathbb{R}$ defining the *post-expectation of* h, denoted by $\mathbb{X}[h]$ and defined as the following function over states:

$$\mathbb{X}[h](s) = \int h(s')T(s, \mathrm{d}s'). \tag{4}$$

This represents the expected value of h evaluated at the next state, given the current state being s. Computing the symbolic representation of a post-expectation for probabilistic programs and stochastic dynamical models is a core problem in probabilistic verification [24, 25]. Indeed, our theoretical framework builds upon the post-expectation (cf. Eq. (13b)), and our verification procedure uses a symbolic representation of the post-expectation in our program-aware method (cf. Eqs. (17) and (19)), while our program-agnostic method approximates it statistically.

We remark that our model encompasses general probabilistic program loops as well as stochastic dynamical systems with general disturbances. For example, a probabilistic loop with guard condition B and body C in the form

$$\texttt{while } B \texttt{ do } C \texttt{ od} \tag{5}$$

can be expressed as a loop with guard `true` and body `if` B `then` C `else skip fi`. This expresses the fact that, after termination, the program will stay on the terminal state indefinitely. Also, discrete-time stochastic difference equations with a nonlinear vector field g and time-invariant input disturbance with an arbitrary distribution $\mathcal{W}$ in the form

$$s_{t+1} = g(s_t, w_t), \quad w_t \sim \mathcal{W}. \tag{6}$$

comply with our model. It is sufficient to derive w_t with an appropriate inverse transformation from the uniform distribution and embed it in f. Our model is even more general, as it encompasses state-dependent distributions, whose parameters depend on the state and may depend on other distributions and thus define joint, multi-variate and hierarchically-structured distributions. Notably, our model comprises both continuous and discrete probability distributions and is able to model discrete-time stochastic hybrid systems.

3 Quantitative Reachability Verification of Probabilistic Models

Quantitative verification treats the question of providing a quantity for which a system satisfies a property as opposed to providing a definite positive or negative answer. In fact, it is sometimes too conservative or inappropriate to demand a definite outcome to a formal verification question. For instance, a system whose behaviour is probabilistic may violate a specification on rare corner cases and yet satisfy it with a probability that is deemed acceptable for the application domain. We address the quantitative verification of probabilistic systems, which is the problem of computing the probability for which a system satisfies a specification [7, 8, 32, 45].

We consider the *quantitative reachability verification* question, which as we show below, is at the core of a variety of quantitative verification questions for probabilistic programs and stochastic dynamical models. Henceforth, we use $\mathbf{1}_S$ to denote the indicator function of set S, i.e., $\mathbf{1}_S(s) = 1$ if $s \in S$ and $\mathbf{1}_S(s) = 0$ if $s \notin S$; we also use $\lambda x.M$ to denote the anonymous function that takes an argument x and evaluates the expression M to produce its result. We now characterise the probability that a stochastic process reaches a Borel measurable target set A in exactly time t, in at most time t, and in any finite time.

▶ **Lemma 1.** *Let the event that a stochastic process $\{X_t^s(\omega)\}_{t\in\mathbb{N}}$ over state space $\mathbb{R}^n$ initialised in state $s \in \mathbb{R}^n$ reaches a target set $A \in \mathcal{B}(\mathbb{R}^n)$ in exactly time $t \in \mathbb{N}$ be*

$$\mathrm{Reach}_t^s(A) = \big\{\omega \in \Omega \mid X_0^s(\omega) \notin A, \ldots, X_{t-1}^s(\omega) \notin A, X_t^s(\omega) \in A\big\}, \tag{7}$$

with $\mathrm{Reach}_0^s(A) = \Omega$ *if $s \in A$, and* $\mathrm{Reach}_0^s(A) = \emptyset$ *if $s \notin A$. Then,* $\mathrm{Reach}_t^s(A)$ *is measurable and its probability measure can be expressed as follows:*

$$\mathbb{P}[\mathrm{Reach}_{t+1}^s(A)] = \mathbf{1}_{\mathbb{R}^n\setminus A}(s) \cdot \mathbb{X}[\lambda s'.\, \mathbb{P}[\mathrm{Reach}_t^{s'}(A)]](s), \tag{8a}$$

$$\mathbb{P}[\mathrm{Reach}_0^s(A)] = \mathbf{1}_A(s). \tag{8b}$$

▶ **Lemma 2.** *Let the event that a stochastic process $\{X_t^s(\omega)\}_{t\in\mathbb{N}}$ over state space $\mathbb{R}^n$ initialised in state $s \in \mathbb{R}^n$ reaches a target set $A \in \mathcal{B}(\mathbb{R}^n)$ in at most time $t \in \mathbb{N}$ be*

$$\mathrm{Reach}_{\leq t}^s(A) = \cup\{\ \mathrm{Reach}_i^s(A) \colon 0 \leq i \leq t\ \}, \tag{9}$$

Then, $\mathrm{Reach}_{\leq t}^s(A)$ *is measurable and its probability measure can be expressed as follows:*

$$\mathbb{P}[\mathrm{Reach}_{\leq t+1}^s(A)] = \mathbf{1}_A(s) + \mathbf{1}_{\mathbb{R}^n\setminus A}(s) \cdot \mathbb{X}[\lambda s'.\, \mathbb{P}[\mathrm{Reach}_{\leq t}^{s'}(A)]](s), \tag{10a}$$

$$\mathbb{P}[\mathrm{Reach}_{\leq 0}^s(A)] = \mathbf{1}_A(s). \tag{10b}$$

▶ **Lemma 3.** *Let the event that a stochastic process $\{X_t^s(\omega)\}_{t\in\mathbb{N}}$ over state space $\mathbb{R}^n$ initialised in state $s \in \mathbb{R}^n$ reaches a target set $A \in \mathcal{B}(\mathbb{R}^n)$ in finite time be*

$$\mathrm{Reach}_{\mathrm{fin}}^s(A) = \cup\{\mathrm{Reach}_t^s(A) \colon t \in \mathbb{N}\} = \cup\big\{\mathrm{Reach}_{\leq t}^s(A) \colon t \in \mathbb{N}\big\}. \tag{11}$$

Then, $\mathrm{Reach}^{s}_{\mathrm{fin}}(A)$ *is measurable and its probability measure (which we refer to as the reachability probability of the target set) can be expressed as follows:*

$$\mathbb{P}[\mathrm{Reach}^{s}_{\mathrm{fin}}(A)] = \lim_{t\to\infty} \mathbb{P}[\mathrm{Reach}^{s}_{\leq t}(A)]. \tag{12}$$

The lemmata above follow from measure-theoretic results in probabilistic verification (proofs are provided in the extended version [2]), and underpin the formal characterisation of quantitative probabilistic reachability in the next section. Our method leverages neural networks to compute an upper bound for the reachability probability with respect to a given target set (cf. Section 4). Notice that an appropriate choice of target set allows us to express a variety of other quantitative verification questions, for which our method can provide an upper or lower bound. Below, by providing a suitable choice for the target set, we show how important quantitative verification questions for probabilistic programs and stochastic dynamical models can be characterised as instances of quantitative reachability.

Termination Analysis. Let $G \in \mathcal{B}(\mathbb{R}^n)$ be the guard set of a probabilistic while loop as in Eq. (5), i.e., the set of states for which that guard condition evaluates to true. The event that the loop terminates from initial state s_0 is $\mathrm{Reach}^{s_0}_{\mathrm{fin}}(\mathbb{R}^n \setminus G)$. Our method computes an upper bound for the probability that the loop terminates or, dually, it computes a lower bound for the probability of non-termination. Notably, when this lower bound is greater than 0, then almost-sure termination is refuted.

Assertion-violation Analysis. Let $G \in \mathcal{B}(\mathbb{R}^n)$ be the guard set of a probabilistic while loop and $A \in \mathcal{B}(\mathbb{R}^n)$ be the satisfying set of an assertion placed at the beginning of the loop body. Given initial state s_0, the event that the assertion is eventually violated is $\mathit{Reach}^{s_0}_{\mathrm{fin}}(G \setminus A)$. Our method computes an upper bound for the probability of assertion violation or, dually, a lower bound for its satisfaction. Note that assertions in other positions of the body can be modelled similarly by using additional Boolean variables.

Safety Verification. Let $B \in \mathcal{B}(\mathbb{R}^n)$ be a set of undesirable states in a stochastic dynamical model. The event that the system is safe when initialised in s_0, i.e., it never reaches an undesirable state, is given by $\Omega \setminus \mathit{Reach}^{s_0}_{\mathrm{fin}}(B)$. Our method computes a lower bound for the probability that the system is safe, which is $1 - \mathbb{P}[\mathit{Reach}^{s_0}_{\mathrm{fin}}(B)]$.

Invariant Verification. Let $I \in \mathcal{B}(\mathbb{R}^n)$ be a candidate invariant set. The event that I is invariant when the system is initialised in s_0 is $\Omega \setminus \mathit{Reach}^{s_0}_{\mathrm{fin}}(\mathbb{R}^n \setminus I)$. Our method computes a lower bound for the probability that set I is invariant, which is $1 - \mathbb{P}[\mathit{Reach}^{s_0}_{\mathrm{fin}}(\mathbb{R}^n \setminus I)]$. Note that if $s_0 \notin I$, this definition yields a trivial lower bound of zero for the probability of invariance.

4 Neural Supermartingales for Quantitative Verification

Supermartingale certificates provide a flexible and powerful theoretical framework for the formal verification of probabilistic models with infinite state spaces. Not only have supermartingales been applied to qualitative questions, such as almost-sure termination analysis of probabilistic programs and almost-sure stability analysis of stochastic dynamical models, but also to quantitative reachability verification, as in this work. Specifically, this is enabled by the theory of non-negative repulsing supermartingales and of stochastic invariants [17, 19, 43].

Our method builds upon the theory of non-negative repulsing supermartingales and stochastic invariant indicators and adapts it to take advantage of the expressive power of neural networks and the flexibility of machine learning and counterexample-guided inductive synthesis algorithms. Our method hinges on the following theorem, whose proof we provide in the extended version [2].

```
while x > 0 do
    assert(x <= 10);
    p ~ Bernoulli(0.5);
    if p == 1 then
        x -= 2
    else
        x += 1
    fi
od
```

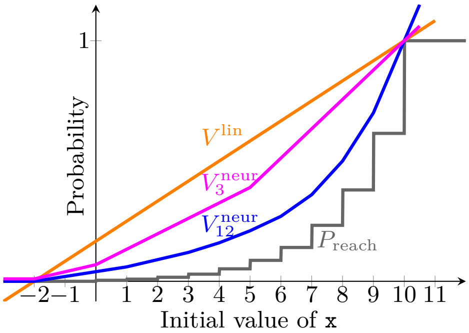


Figure 2 Comparison between linear and neural supermartingale functions in tightness of bounds for the assertion violation probability of the program repulse, shown on the left. On the right, the function V^{lin} indicates the tightest linear supermartingale (under the restriction that x is greater than -2). The functions V_3^{neur} and V_{12}^{neur} indicate single-layer neural supermartingales with 3 and 12 neurons respectively. The piecewise constant function P_{reach} is the true probability of assertion violation, and indicates the ideal lower bound for the value of any supermartingale function.

▶ **Theorem 4.** *Let $\mathbb{X}$ be the post-expectation operator of a stochastic process over state space $\mathbb{R}^n$ and let $A \in \mathcal{B}(\mathbb{R}^n)$ be a target set. Let $V : \mathbb{R}^n \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ be a non-negative function that satisfies the following two conditions:*

$$\textit{(indicating condition)} \qquad \forall s \in A\colon V(s) \geq 1, \tag{13a}$$

$$\textit{(non-increasing condition)} \qquad \forall s \notin A\colon \mathbb{X}[V](s) \leq V(s), \tag{13b}$$

Then, for every state $s \in \mathbb{R}^n$, it holds that $V(s) \geq \mathbb{P}[\text{Reach}^s_{\text{fin}}(A)]$.

As we show in detail in Section 5, we compute a supermartingale that satisfies the two criteria (13a) and (13b), while also minimising its output over the initial state s_0. When the initial state is chosen nondeterministically from a set S_0, we instead minimise the output over all states in the set. As a consequence, the maximum of V over S_0 is a sound upper bound for the reachability probability.

In this work, we template V as a neural network with n input neurons, l hidden layers with respectively $h_1, \ldots h_l$ neurons in each hidden layer, and 1 output neuron. We guarantee a-priori that the function's output will be non-negative over the entire domain using the following architecture:

$$V(x) = \text{sum}\left(\sigma_l \circ \cdots \circ \sigma_1(x)\right), \quad \sigma_i(z) = (\text{ReLU}(\mathbf{w}^T_{i,1} z + \mathbf{b}_{i,1}), \ldots, \text{ReLU}(\mathbf{w}^T_{i,h_i} z + \mathbf{b}_{i,h_i})) \tag{14}$$

where $\text{sum}(z_{l,1}, \ldots, z_{l,h_l}) = \sum_{k=1}^{h_l} z_{l,k}$ and $\mathbf{w}^T_{i,j} \in \mathbb{Q}^{h_{i-1}}$ (defining $h_0 = n$, the number of input neurons) and $\mathbf{b}_{i,j} \in \mathbb{Q}$ are respectively the weight vector and the bias parameter for the inputs to neuron j at layer i. Then, our method trains the neural network to satisfy the two criteria (13a) and (13b), while also minimising its output on the initial condition.

Using neural networks as templates of supermartingale functions introduces non-trivial advantages with respect to symbolic methods for the synthesis of supermartingales. Firstly, neural supermartingales are able to better approximate the true probability of reachability and thus attain tighter upper bounds on it. Secondly, symbolic methods require deterministic invariants that overapproximate the set of reachable states, and suitably restricts the domain of the template. Figure 2 illustrates using an example the advantages of neural certificates with respect to linear supermartingales synthesised using Farkas' lemma. This example shows that increasing the number of neurons provides greater flexibility and allows the certificate

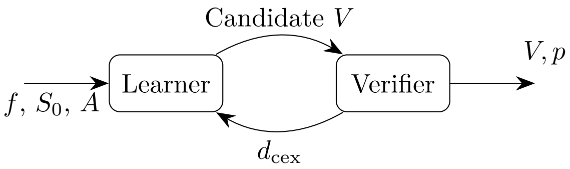


Figure 3 Overview of the counterexample-guided inductive synthesis procedure used to synthesise neural supermartingales for quantitative reachability verification. Inputs to the procedure are a probabilistic program f, a set of initial states S_0, and a target set A. The procedure outputs a valid neural supermartingale V and a probability bound p.

to more tightly approximate the true probability. Moreover, this example shows that linear supermartingales require their domain to be restricted with an appropriate deterministic invariant. Notably, symbolic methods for the synthesis of polynomial supermartingales based on Putinar's Positivstellensatz also require compact deterministic invariants to be provided [17]. By contrast, neural supermartingales (whose output is always non-negative) achieve the same result while relaxing the requirement of providing an invariant beforehand.

5 Data-driven Synthesis of Neural Supermartingales

Our approach to synthesising neural supermartingales for quantitative verification utilises a counterexample-guided inductive synthesis (CEGIS) procedure [41, 42] (cf. Figure 3). This procedure consists of two components, a learner and a verifier, that work in opposition to each other. On the one hand, the learner seeks to synthesise a candidate supermartingale that meets the desired specification (cf. Eq. (13)) over a finite set of samples from the state space, while simultaneously optimising the tightness of the probability bound. On the other hand, the verifier seeks to disprove the validity of this candidate by searching for counterexamples, i.e., instances where the desired specification is invalidated, over the entire state space. If the verifier shows that no such counterexample exists, then the desired specification is met by the supermartingale and the procedure provides a sound probability bound for the reachability probability of interest, together with a neural supermartingale to certify it.

5.1 Training of Neural Supermartingales From Samples

Our neural supermartingale for quantitative reachability verification consists of a neural network with ReLU activation functions, with an arbitrary number of hidden layers (cf. Section 4). We train this neural network using gradient descent over a finite set $D = \{d^{(1)}, \ldots, d^{(m)}\} \subseteq \mathbb{R}^n$ of states $d^{(i)}$ sampled over the state space $\mathbb{R}^n$. Initially, we sample uniformly within a bounded hyper-rectangle of $\mathbb{R}^n$, a technique that scales effectively to high dimensional state spaces and efficiently populates the initial dataset of state samples. Then, we construct a loss function that guides gradient descent to optimise the parameters (weight and biases) of the neural network V to satisfy the specification set out in Eq. (13) while also minimising the probability bound. We define a loss function $\mathcal{L}(D)$ that consists of three terms:

$$\mathcal{L}(D) = \beta_1 \mathcal{L}_{\text{ind}}(D) + \beta_2 \mathcal{L}_{\text{non-inc}}(D) + \beta_3 \mathcal{L}_{\text{min}}(D). \tag{15}$$

Components $\mathcal{L}_{\text{ind}}$ and $\mathcal{L}_{\text{non-inc}}$ are responsible for encouraging satisfaction of the conditions in Eq. (13), while the component $\mathcal{L}_{\text{min}}$ is responsible for tightening the probability bound. The parameters of this optimisation problem are the parameters of the neural network, which are initialised randomly. The dataset consists of the state samples, initially sampled randomly, and generated from counterexamples in subsequent CEGIS iterations (cf. Section 5.2). The coefficients β_1, β_2 and β_3 denote scale factors for each term, which we choose according to the priority that we want to assign to each condition (cf. Section 6).

First, consider the condition in Eq. (13a), which we refer to as the *indicating condition*. For this, we use the following loss function:

$$\text{(indicating loss)} \qquad \mathcal{L}_{\text{ind}}(D) = \mathbb{E}_{d \in D \cap A}[\text{ReLU}(1 - V(d))]. \tag{16}$$

This adds a penalty for states $d \in D$ lying inside the target condition A, at which V fails to satisfy the indicating condition, whilst ignoring any states where V satisfies it. We average this per-state penalty across all states in $D \cap A$.

We next consider the *non-increasing condition* in Eq. (13b), for which we use the following loss term:

$$\text{(non-increasing loss)} \qquad \mathcal{L}_{\text{non-inc}}(D) = \mathbb{E}_{d \in D \setminus A}[\text{ReLU}(\mathbb{X}[V](d) - V(d))]. \tag{17}$$

This penalises states $d \in D$ lying outside of the target set A, at which V fails to satisfy the non-increasing condition. Notably, this component is defined in terms of the post-expectation $\mathbb{X}[V]$ of our supermartingale. To embed this expression in our loss function we consider two alternative approaches, which we call *program-aware* and *program-agnostic*.

Program-aware Approach. The program-aware approach uses the source code of the program to generate a symbolic expression for the post-expectation of V. For this purpose, we exploit the symbolic inference algorithm introduced by the tool PSI [24, 25], along with a symbolic representation of V. We construct a probabilistic program which represents the evaluation of V on the state resulting after the execution of the update function f. The expected value of this program is precisely $\mathbb{X}[V]$. This results in a symbolic expression that is a function of the program state and parameters of V. In the non-increasing loss, states are instantiated to elements of $D \setminus A$, while the parameters are left as free-variables that the gradient descent engine differentiates with respect to.

Program-agnostic Approach. The program-agnostic approach provides an alternative formulation of the non-increasing loss term that does not require symbolic reasoning. Instead of leveraging the program's source code, it only requires the ability to execute it. For this, we utilise a Monte Carlo scheme to estimate the post-expectation. For each state d in our dataset D, to obtain an estimate of $\mathbb{X}[V](d)$ we sample a number m' of successor states $D' = \{d'^{(k)} \ : \ 1 \leq k \leq m'\}$. Each successor state $d'^{(k)}$ is sampled by executing the program's update function f (cf. Eq. (1)) at state d. Then $\mathbb{X}[V](d)$ is estimated as $\mathbb{X}[V](d) \approx \mathbb{E}_{d' \in D'}[V(d')]$ which is the average of V over D'. Even though this is an approximation, we emphasise that this does not affect the soundness of our scheme, which is ensured by the verifier.

Finally, we introduce a tightness criterion that minimises the probability upper bound:

$$\text{(minimisation loss)} \qquad \mathcal{L}_{\text{min}}(D) = \mathbb{E}_{d \in D \cap S_0}[V(d)]. \tag{18}$$

This term encourages V to take smaller values over the set of initial states S_0. Recall that the smaller the probability upper bound, the closer it is to the true value.

The loss function is provided to the gradient descent optimiser, whose performance benefits from a smooth objective. For this reason, to improve the performance of our learner we replace every ReLU with a smooth approximation, Softplus, which takes the form $\text{Softplus}(s) = \log(1 + \exp(s))$. Additionally, we improve the approximation of Softplus to ReLU at small values over the interval $[0, 1]$ by re-scaling V. This means that we modify the bounding condition to require that $V(x) \geq \alpha$ over the target set A, for some large $\alpha > 1$. In other words, we modify the indicating loss to $\mathcal{L}_{\text{ind}}(V) = \mathbb{E}_{d \in D \cap A}[\text{ReLU}(\alpha - V(d))]$. We remark that while Softplus is used as the activation function in the learning stage, for the verification stage we instead employ ReLU activation functions, ensuring soundness of the generated neural supermartingale. We remark that this has no effect on the soundness of our approach, which is ultimately guaranteed by the verifier.

5.2 Verification of Neural Supermartingales Using SMT Solvers

The purpose of the verification stage is to check that the neural supermartingale meets the requirements of Eq. (13) over the entire state space $\mathbb{R}^n$, and if that is determined to be the case to furthermore obtain a sound upper bound on the reachability probability. We achieve this by constructing a suitable formula in first-order logic, Eq. (19), and use SMT solving to decide its validity, or equivalently, to decide the satisfiability of its negation, Eq. (20).

The conditions pertaining to the validity of the neural supermartingale, given in (13a) and (13b), are encoded by the formulas φ_{ind} and $\varphi_{\text{non-inc}}$. We also note that for a constant $p \in [0, 1]$ to be a sound upper bound on the reachability probability, it is sufficient to require that p is an upper bound on the neural supermartingale's value over the set of initial states S_0 (cf. Theorem 4), which is expressed by the formula φ_{bound}. A suitable choice for the bound p is determined by a binary search over the interval $[0, 1]$.

$$\forall s \in \mathbb{R}^n : \underbrace{(s \in A \to V(s) \geq 1)}_{\varphi_{\text{ind}}} \wedge \underbrace{(s \notin A \to \mathbb{X}[V](s) \leq V(s))}_{\varphi_{\text{non-inc}}} \wedge \underbrace{(s \in S_0 \to V(s) < p)}_{\varphi_{\text{bound}}}. \tag{19}$$

Here, $V(s)$ is a symbolic encoding of the candidate neural supermartingale proposed by the learner (Section 5.1), and S_0 and A are defined by Boolean predicates over program variables, all of which (in our setting of networks composed from ReLU activations) are expressible using expressions and constraints in non-linear real arithmetic. For this reason, we use Z3 as our SMT solver [21].

The SMT solver is provided with the negation of Eq. (19), namely

$$\exists s \in \mathbb{R}^n : \underbrace{(s \in A \wedge V(s) < 1)}_{\neg\varphi_{\text{ind}}} \vee \underbrace{(s \notin A \wedge \mathbb{X}[V](s) > V(s))}_{\neg\varphi_{\text{non-inc}}} \vee \underbrace{(s \in S_0 \wedge V(s) \geq p)}_{\neg\varphi_{\text{bound}}}, \tag{20}$$

and decides its satisfiability, seeking an assignment d_{cex} of s that is a counterexample to the neural supermartingale's validity, for which any of $\neg\varphi_{\text{ind}}$, $\neg\varphi_{\text{non-inc}}$ and $\neg\varphi_{\text{bound}}$ are satisfied. If no counterexample is found, this certifies the validity of the neural supermartingale. Alternatively, if a counterexample d_{cex} is found, it is added to the data set D, for the synthesis to incrementally resume.

6 Experimental Evaluation

The previous section develops a method for synthesising neural supermartingales. This section presents an empirical evaluation of the method, by testing it against a series of benchmarks. Each benchmark is tested ten times. A test is successful if a valid supermartingale is synthesised, and the proportion of successful tests is recorded. Further, the average bound from the valid supermartingales is also recorded, along with the average time taken by the learning and verification steps, respectively. This procedure is applied separately for program-aware and program-agnostic synthesis. We use values $\beta_1 = 10, \beta_2 = 10^5, \beta_3 = 1$ in Eq. (15) to define the priority ordering of the three terms in the loss function, which we find beneficial across our set of benchmarks. To compare our method against existing work, we

Table 1 Results comparing neural supermartingales with Farkas Lemma for different benchmarks. Here, p is the average probability bound generated by the certificate; success ratio is the number of successful experiments, out of 10 repeats, generated by CEGIS with neural supermartingale; '-' means no result was obtained. We also denote the architecture of the network: (h_1, h_2) denotes a network with 2 hidden layers consisting of h_1 and h_2 neurons respectively.

Benchmark	**Farkas'**	**Quantitative Neural Certificates**				**Network**
	Lemma	Program-Agnostic		Program-Aware		**Arch.**
		p	Success Ratio	p	Success Ratio	
`persist_2d`	-	≤ 0.1026	0.9	≤ 0.1175	0.9	(3, 1)
`faulty_marbles`	-	≤ 0.0739	0.9	≤ 0.0649	0.8	3
`faulty_unreliable`	-	≤ 0.0553	0.9	≤ 0.0536	1.0	3
`faulty_regions`	-	≤ 0.0473	0.9	≤ 0.0411	0.9	(3, 1)
`cliff_crossing`	≤ 0.4546	≤ 0.0553	0.9	≤ 0.0591	0.8	4
`repulse100`	≤ 0.0991	≤ 0.0288	1.0	≤ 0.0268	1.0	3
`repulse100_uniform`	≤ 0.0991	≤ 0.0344	1.0	-	-	2
`repulse100_2d`	≤ 0.0991	≤ 0.0568	1.0	≤ 0.0541	1.0	3
`faulty_varying`	≤ 0.1819	≤ 0.0864	1.0	≤ 0.0865	1.0	2
`faulty_concave`	≤ 0.1819	≤ 0.1399	1.0	≤ 0.1356	0.9	(3, 1)
`fixed_loop`	≤ 0.0091	≤ 0.0095	1.0	≤ 0.0094	1.0	1
`faulty_loop`	≤ 0.0181	≤ 0.0195	1.0	≤ 0.0184	1.0	1
`faulty_uniform`	≤ 0.0181	≤ 0.0233	1.0	≤ 0.0221	1.0	1
`faulty_rare`	≤ 0.0019	≤ 0.0022	1.0	≤ 0.0022	1.0	1
`faulty_easy1`	≤ 0.0801	≤ 0.1007	1.0	≤ 0.0865	1.0	1
`faulty_ndecr`	≤ 0.0561	≤ 0.0723	1.0	≤ 0.0630	1.0	1
`faulty_walk`	≤ 0.0121	≤ 0.0173	1.0	≤ 0.0166	1.0	1

perform template-based synthesis of linear supermartingales using Farkas' Lemma. This requires deterministic invariants to overapproximate the reachable set of states, which may either be generated by abstract interpretation, or provided manually [43]. In our experiments, we provide a suitable invariant manually based on the guard of the loop, in some cases strengthening them with additional constraints by an educated guess.

It should be noted that our method is inherently stochastic. One reason is the random initialisation of the neural template's parameters in the learning phase. In program-agnostic synthesis, an additional source of randomness is the sampling of successor states. So that the results accurately reflect the performance of our method, the random seed for these sources of randomness is selected differently for each test. An additional source of non-determinism arises from the SMT solver Z3 as it generates counterexamples: this cannot be controlled externally. Benchmarks are run on a machine with an Nvidia A40 GPU, and involve the assertion-violation analysis of programs created using the following two patterns:

Unreliable Hardware. These are programs that execute on unreliable hardware. The goal is to upper bound the probability that the program fails to terminate due to a hardware fault. A simple example is `faulty_loop`, whose source code is presented in the extended version [2], and which consists of a loop which may violate an assertion with small probability, modelling a hardware fault.

Robot Motion. These programs model an agent (e.g., a robot) that moves within a physical environment. In these benchmarks, the uncertainty in control and sensing is modelled probabilistically. The environment contains a target region and a hazardous region. The goal is to upper bound the probability that the robot enters the hazardous region. We provide the program `repulse100` as an example, which is variant of `repulse` (Figure 2)

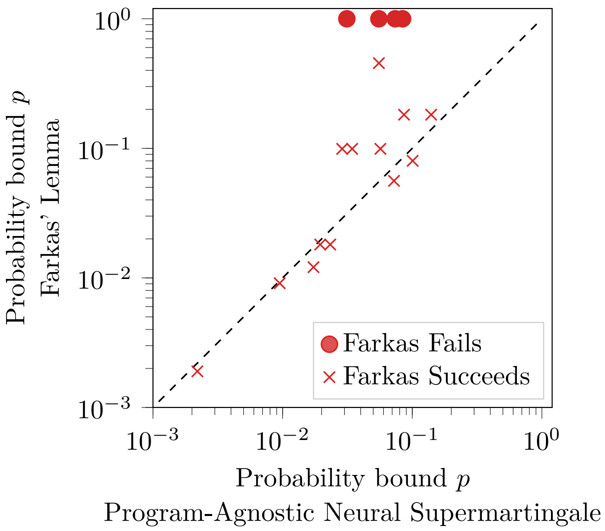


Figure 4 Probability bounds generated using program-agnostic neural supermartingales and using Farkas' Lemma. The reference line $y = x$ allows one to see which approach outperforms the other and by how much: above the line means that neural supermartingales outperform Farkas' Lemma, below the line the opposite. Neural supermartingales can significantly outperform linear templates when a better bound exists, but otherwise achieve similar results. Our approach with program-aware neural supermartingales provides even better outcomes, compared with Farkas' Lemma.

containing a modified assertion and initial state. The program models the motion of a robot in a one-dimensional environment, starting at $\texttt{x} = 10$. The target region is where $\texttt{x} < 0$, and hazardous region is where $\texttt{x} > 100$. As with `repulse`, in each iteration there is an equal probability of `x` being decremented by 2, and x being incremented by 1.

Several of the programs are based on benchmarks used in prior work focusing on other types of supermartingales [6, 13]. Additionally, there are several benchmarks that are entirely new.

The results are reported in Table 1. The table is divided into three sections. The first section shows the benchmarks where Farkas' Lemma cannot be applied, and where only our method is capable of producing a bound. The second section shows examples where both methods are able to produce a bound, but our method produces a notably better bound. The third section shows benchmarks where both methods produce comparable bounds.

Dashes in the table indicate experiments where a valid supermartingale could not be obtained. In the case of Farkas' Lemma, there are several cases where there is no linear supermartingale for the benchmark. By contrast, program-agnostic and program-aware synthesis could be applied to almost all benchmarks. The one exception is `repulse100_uniform` where only program-aware verification was unsuccessful: this is due to indicator functions in the post-expectation, which are not smooth and posed a problem for the optimiser. This benchmark underscores the value of program-agnostic synthesis, since it does not require embedding the explicit post-expectation in the loss function.

The first section of the benchmarks in Table 1 demonstrates that our method produces useful results on programs that are out-of-scope for existing techniques. Furthermore, the success ratio of our method is high on all the benchmarks, which indicates its robustness. For the second and third sections (which consist of benchmarks to which Farkas' Lemma is applicable), the success ratio of our method is broadly maximal, which is to be expected, since these programs can also be solved by Farkas' Lemma.

Table 2 Results showing the time taken in seconds to synthesise supermartingales by our method and Farkas' Lemma. For our method, we show the time taken during learning and verification.

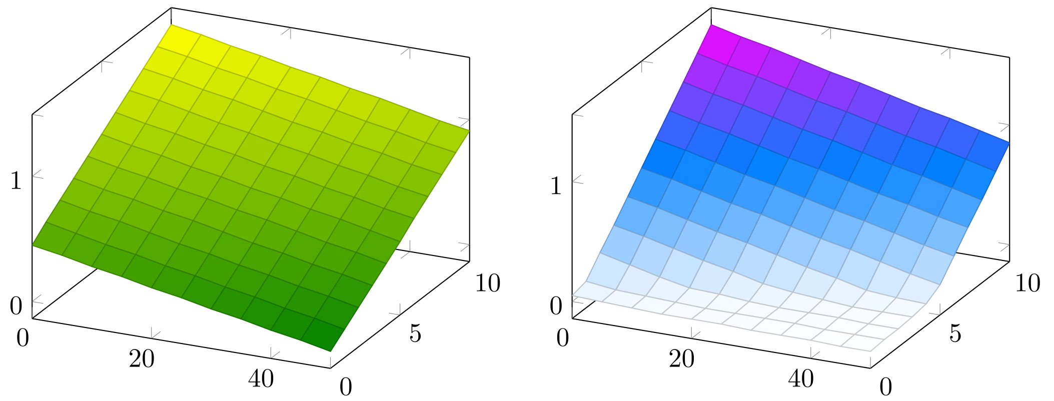
Benchmark	**Farkas'**	**Quantitative Neural Certificates**			
	Lemma	Program-Agnostic		Program-Aware	
		Learn Time	Verify Time	Learn Time	Verify Time
`persist_2d`	-	169.14	85.31	44.96	74.90
`faulty_marbles`	-	114.24	29.23	15.86	28.68
`faulty_unreliable`	-	123.85	45.48	18.34	33.97
`faulty_regions`	-	17.92	35.85	17.55	32.38
`cliff_crossing`	0.11	134.61	19.02	21.27	29.07
`repulse100`	0.19	16.65	5.00	6.49	3.74
`repulse100_uniform`	0.19	21.28	14.18	-	-
`repulse100_2d`	0.12	122.92	64.54	15.75	47.70
`faulty_varying`	0.36	21.74	5.06	4.71	3.28
`faulty_concave`	0.39	49.12	13.37	13.49	7.82
`fixed_loop`	0.15	14.16	3.14	3.34	2.43
`faulty_loop`	0.16	25.52	3.81	3.73	2.66
`faulty_uniform`	0.34	20.20	1.91	6.75	1.33
`faulty_rare`	0.27	25.52	4.27	3.71	2.96
`faulty_easy1`	0.31	104.20	12.78	4.95	7.51
`faulty_ndecr`	0.33	104.89	9.06	5.37	4.66
`faulty_walk`	0.32	15.08	4.00	6.97	3.33

In the second section of Table 1, we find more complex benchmarks where our method was able to significantly improve the bound from Farkas' Lemma. The smallest improvement was about 0.04, and the largest improvement was over 0.39. The intuition here is that neural templates allow more sophisticated supermartingales to be learnt, that can approximate how the reachability probability varies across the state space better than linear templates, and thereby yield tighter probability bounds.

The third section of Table 1 consists of relatively simple benchmarks, where our method produces results that are marginally less tight in comparison to Farkas' Lemma. This is not surprising since our method uses neural networks consisting of a single neuron for these examples, owing to their simplicity. The expressive power of these networks is therefore similar to linear templates.

In summary, the results show that our method does significantly better on more complex examples, and marginally worse on very simple examples. This is highlighted in Figure 4. Each point represents a benchmark. The position on the x-axis shows the probability bound obtained by our program-agnostic method, and the y-axis shows the probability bound obtained by Farkas' Lemma. Points above the line indicate benchmarks where neural supermartingales outperform Farkas' Lemma, and vice versa. The scale is logarithmic to emphasise order-of-magnitude differences.

Notice that in Table 1 the program-aware algorithm usually yields better bounds than the program-agnostic algorithm, but the improvement is mostly marginal. This is in fact a strength of our method: our data-driven approach performs almost as well as one dependent on symbolic representations, which is promising in light of questions of scalability to more complex programs. We also include a breakdown of computation time (Table 2) which allows distinguishing between learning and verification overheads. Notably, Farkas' Lemma



Linear supermartingale.

Neural supermartingale.

Figure 5 Supermartingale functions for the `cliff_crossing` benchmark as generated using Farkas' Lemma (on the left) and using neural supermartingales (on the right). The right hand figure illustrates the tighter bounds obtainable through the use of neural templates.

is significantly faster than our method, given that it relies on solving a convex optimisation problem via linear programming, whereas the synthesis of neural supermartingales is a non-convex optimisation problem that is addressed using gradient descent.

Having presented the experimental results, we shall further comment on some specific benchmarks. The `repulse100` program in Table 1 is a variation of `repulse` presented in Figure 2, but with the assertion changed to `assert(x <= 100)`. While this is a small program, our method is still able to produce a significantly better result than Farkas' lemma, using a neural supermartingale with a single hidden layer consisting of three ReLU components that are summed together, which allows a convex piecewise linear function to be learnt. The `cliff_crossing` program is a further benchmark for which our method is capable of producing a significantly better bound. This is a 2 dimensional benchmark, for which we use a neural supermartingale that consists of two input neurons and four ReLU components, leading to a clear improvement compared to the linear supermartingale in the tightness of the probability bounds generated, as illustrated by Figure 5. Both `repulse100` and `cliff_crossing` are benchmarks that use neural supermartingales with a single hidden layer. An example that uses two hidden layers is `faulty_concave`, in which there are two distinct regions of the state space, one of which has a significantly higher reachability probability than the other. We find that neither a linear template nor a single-layer neural supermartingale is able to exploit this conditional behaviour, each of which yield an overly conservative certificate, but that a neural supermartingale with two hidden layers is able to more tightly approximate the reachability probability in each of the two regions. Further discussion and the source code of these case studies are presented in the extended version [2].

7 Related Work

The formal verification of probabilistic programs using supermartingales is a well-studied topic. Early approaches to introduce this technique applied them to almost-sure termination analysis of probabilistic programs [13], which allowed several extensions to polynomial programs, programs with non-determinism, lexicographic and modular termination arguments, and persistence properties [5, 15, 16, 19, 23, 29]. All these methods relied on symbolic reasoning algorithms for synthesising supermartingales, that leveraged theories based on Farkas' lemma for the synthesis of linear certificates, and Putinar's Positivestellensatz and sum-of-square methods for the synthesis of polynomial certificates. While these methods are the state-of-the-art for many existing problem instances in literature, to achieve the strong guarantees that they provide (such as completeness for the specific class of programs they target), they must necessarily introduce restrictions on the class of programs to which they are applicable, and the form of certificates that they derive. Moreover, symbolic methods need externally provided invariants that are stronger than $\mathbb{R}^n$ to enforce non-negativity in the case of linear certificates, as we illustrate in Figure 2. Also, symbolic methods for the synthesis of polynomial certificates require compact deterministic invariants to operate.

The use of neural networks to represent certificates has allowed many of these restrictions to be lifted. In the context of the analysis of probabilistic programs, neural networks were first applied to certify positive almost-sure termination [3]. This approach lent itself to a wider range of formal verification questions for stochastic dynamical models, from stability and safety analysis to controller synthesis [18, 34, 35]. These data-driven inductive synthesis techniques for supermartingales have also been extended to machine learning techniques other then deep learning, such as piecewise linear regression and decision tree learning [10, 11].

In this paper, we further extend data-driven synthesis of neural supermartingale certificates to quantitative verification questions. The correctness of our approach builds upon the theory of non-negative repulsing supermartingales [43], as formulated in Theorem 4. Our experiments have demonstrated that neural certificates attain comparable results on programs that are amenable to symbolic analysis (such as those in the third section of Table 1), while surpassing symbolic methods on more complex programs that are either out-of-scope or yield overly conservative bounds when existing techniques are applied (such as those in the first and second section of Table 1).

8 Conclusion

We have presented a data-driven framework for the quantitative verification of probabilistic models that leverage neural networks to represent supermartingale certificates. Our experiments have shown that neural certificates are applicable to a wider range of probabilistic models than was previously possible using purely symbolic techniques. We also illustrate that on existing models our method yields certificates of better or comparable quality than those produced by symbolic techniques for the synthesis of linear supermartingales. This builds upon the ability of neural networks to approximate non-linear functions, while satisfying the constraints imposed by Theorem 4 without the need for supporting deterministic invariants to be provided externally. Our method applies to quantitative termination and assertion-violation analysis for probabilistic programs, as well as safety and invariant verification for stochastic dynamic models. We imagine extensions to further quantitative verification questions, such as temporal properties beyond reachability [9], and bounding expected accrued costs [39, 46, 47].

References

1 Alessandro Abate, Daniele Ahmed, Mirco Giacobbe, and Andrea Peruffo. Formal synthesis of Lyapunov neural networks. *IEEE Control. Syst. Lett.*, 5(3):773–778, 2021.

2 Alessandro Abate, Alec Edwards, Mirco Giacobbe, Hashan Punchihewa, and Diptarko Roy. Quantitative verification with neural networks, 2023. `arXiv:2301.06136`.

3 Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. Learning probabilistic termination proofs. In *CAV (2)*, volume 12760 of *Lecture Notes in Computer Science*, pages 3–26. Springer, 2021.

4 Alessandro Abate, Joost-Pieter Katoen, and Alexandru Mereacre. Quantitative automata model checking of autonomous stochastic hybrid systems. In *HSCC*, pages 83–92. ACM, 2011.

5 Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proc. ACM Program. Lang.*, 2(POPL):34:1–34:32, 2018.

6 Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2010.

7 Christel Baier, Luca de Alfaro, Vojtech Forejt, and Marta Kwiatkowska. Model checking probabilistic systems. In *Handbook of Model Checking*, pages 963–999. Springer, 2018.

8 Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Performance evaluation and model checking join forces. *Commun. ACM*, 53(9):76–85, 2010.

9 Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

10 Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. Data-driven invariant learning for probabilistic programs. In *CAV (1)*, volume 13371 of *Lecture Notes in Computer Science*, pages 33–54. Springer, 2022.

11 Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. Probabilistic program verification via inductive synthesis of inductive invariants. In *TACAS (2)*, volume 13994 of *Lecture Notes in Computer Science*, pages 410–429. Springer, 2023.

12 D. P. Bertsekas and S. E. Shreve. *Stochastic optimal control: The discrete-time case.* Athena Scientific, 1996.

13 Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013.

14 Ya-Chien Chang, Nima Roohi, and Sicun Gao. Neural Lyapunov control. In *NeurIPS*, pages 3240–3249, 2019.

15 Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination analysis of probabilistic programs through positivstellensatz's. In *CAV (1)*, volume 9779 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2016.

16 Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. *ACM Trans. Program. Lang. Syst.*, 40(2):7:1–7:45, 2018.

17 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Đorđe Žikelić. Sound and complete certificates for quantitative termination analysis of probabilistic programs. In *CAV (1)*, volume 13371 of *Lecture Notes in Computer Science*, pages 55–78. Springer, 2022.

18 Krishnendu Chatterjee, Thomas A. Henzinger, Mathias Lechner, and Đorđe Žikelić. A learner-verifier framework for neural network controllers and certificates of stochastic systems. In *TACAS (1)*, volume 13993 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2023.

19 Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. Stochastic invariants for probabilistic termination. In *POPL*, pages 145–160. ACM, 2017.

20 Fredrik Dahlqvist and Alexandra Silva. Semantics of probabilistic programming: A gentle introduction. In Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva, editors, *Foundations of Probabilistic Programming*, pages 1–42. Cambridge University Press, 2020.

21 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

22 Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Quantitative multi-objective verification for probabilistic systems. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2011.

23 Hongfei Fu and Krishnendu Chatterjee. Termination of nondeterministic probabilistic programs. In *VMCAI*, volume 11388 of *LNCS*, pages 468–490. Springer, 2019.

24 Timon Gehr, Sasa Misailovic, and Martin T. Vechev. PSI: exact symbolic inference for probabilistic programs. In *CAV (1)*, volume 9779 of *Lecture Notes in Computer Science*, pages 62–83. Springer, 2016.

25 Timon Gehr, Samuel Steffen, and Martin T. Vechev. λPSI: exact inference for higher-order probabilistic programs. In *PLDI*, pages 883–897. ACM, 2020.

26 Mirco Giacobbe, Daniel Kroening, and Julian Parsert. Neural termination analysis. In *ESEC/SIGSOFT FSE*, pages 633–645. ACM, 2022.

27 Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *FOSE*, pages 167–181. ACM, 2014.

28 O. Hernández-Lerma and J. B. Lasserre. *Discrete-time Markov control processes*, volume 30 of *Applications of Mathematics*. Springer-Verlag, 1996.

29 Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Modular verification for almost-sure termination of probabilistic programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):129:1–129:29, 2019.

30 O. Kallenberg. *Foundations of modern probability*. Springer Science & Business Media, 2006.

31 Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

32 Marta Z. Kwiatkowska. Quantitative verification: models techniques and tools. In *ESEC/SIGSOFT FSE*, pages 449–458. ACM, 2007.

33 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Quantitative analysis with the probabilistic model checker PRISM. In *QAPL*, volume 153 of *Electronic Notes in Theoretical Computer Science*, pages 5–31. Elsevier, 2005.

34 Mathias Lechner, Đorđe Žikelić, Krishnendu Chatterjee, and Thomas A. Henzinger. Stability verification in stochastic control systems via neural network supermartingales. In *AAAI*, pages 7326–7336. AAAI Press, 2022.

35 Frederik Baymler Mathiesen, Simeon Craig Calvert, and Luca Laurenti. Safety certification for stochastic systems via neural barrier functions. *IEEE Control. Syst. Lett.*, 7:973–978, 2023.

36 Annabelle McIver and Carroll Morgan. Games, probability and the quantitative μ-calculus qmμ. In *LPAR*, volume 2514 of *Lecture Notes in Computer Science*, pages 292–310. Springer, 2002.

37 Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

38 Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996.

39 Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. In *PLDI*, pages 496–512. ACM, 2018.

40 Andrea Peruffo, Daniele Ahmed, and Alessandro Abate. Automated and formal synthesis of neural barrier certificates for dynamical models. In *TACAS (1)*, volume 12651 of *Lecture Notes in Computer Science*, pages 370–388. Springer, 2021.

41 Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, USA, 2008.

42 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

43 Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. Ranking and repulsing supermartingales for reachability in randomized programs. *ACM Trans. Program. Lang. Syst.*, 43(2):5:1–5:46, 2021.

44 Ilya Tkachev, Alexandru Mereacre, Joost-Pieter Katoen, and Alessandro Abate. Quantitative model-checking of controlled discrete-time markov processes. *Inf. Comput.*, 253:1–35, 2017.

45 Franck van Breugel and James Worrell. Towards quantitative verification of probabilistic transition systems. In *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 421–432. Springer, 2001.

46 Di Wang, Jan Hoffmann, and Thomas W. Reps. Central moment analysis for cost accumulators in probabilistic programs. In *PLDI*, pages 559–573. ACM, 2021.

47 Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. Cost analysis of nondeterministic probabilistic programs. In *PLDI*, pages 204–220. ACM, 2019.

Satisfiability Checking of Multi-Variable TPTL with Unilateral Intervals Is PSPACE-Complete

Shankara Narayanan Krishna
IIT Bombay, Mumbai, India

Khushraj Nanik Madnani
MPI-SWS, Kaiserslautern, Germany

Rupak Majumdar
MPI-SWS, Kaiserslautern, Germany

Paritosh Pandya
IIT Bombay, Mumbai, India

Abstract

We investigate the decidability of the $0,\infty$ fragment of Timed Propositional Temporal Logic (TPTL). We show that the satisfiability checking of $\mathrm{TPTL}^{0,\infty}$ is PSPACE -complete. Moreover, even its 1-variable fragment (1-$\mathrm{TPTL}^{0,\infty}$) is strictly more expressive than Metric Interval Temporal Logic (MITL) for which satisfiability checking is EXPSPACE complete. Hence, we have a strictly more expressive logic with computationally easier satisfiability checking. To the best of our knowledge, $\mathrm{TPTL}^{0,\infty}$ is the first multi-variable fragment of TPTL for which satisfiability checking is decidable without imposing any bounds/restrictions on the timed words (e.g. bounded variability, bounded time, etc.). The membership in PSPACE is obtained by a reduction to the emptiness checking problem for a new "non-punctual" subclass of Alternating Timed Automata with multiple clocks called Unilateral Very Weak Alternating Timed Automata ($\mathrm{VWATA}^{0,\infty}$) which we prove to be in PSPACE . We show this by constructing a simulation equivalent non-deterministic timed automata whose number of clocks is polynomial in the size of the given $\mathrm{VWATA}^{0,\infty}$.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases TPTL, Satisfiability, Non-Punctuality, Decidability, Expressiveness, ATA

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.23

Related Version *Full Version*: `https://arxiv.org/abs/2309.00386`

Funding This research was sponsored in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248—CPEC.

Acknowledgements We thank Tom Henzinger for encouraging us to explore non-punctual subclasses for multi-clock TPTL and ATA. We also thank Hsi-Ming Ho for an insightful discussion.

1 Introduction

Metric Temporal Logic ($\mathrm{MTL}[\mathsf{U}_I, \mathsf{S}_I]$) and Timed Propositional Temporal Logic ($\mathrm{TPTL}[\mathsf{U}_I, \mathsf{S}_I]$) are natural extensions of Linear Temporal Logic (LTL) for specifying real-time properties [6]. MTL extends the U and S modality of LTL by associating a time interval with these. Intuitively, $a\mathsf{U}_I b$ is true at a point in the given behaviour iff event a keeps on occurring until at some future time point within relative time interval I, event b occurs. (Similarly, $a\mathsf{S}_I b$ is its mirror image specifying the past behaviour.) On the other hand, TPTL uses freeze quantifiers to store the current time stamp. A freeze quantifier [4, 6] has the form $x.\varphi$ with freeze variable x (also called a clock [7, 27]). When it is evaluated at a point i on a timed word, the time stamp of i (say τ_i) is frozen or registered in x, and the formula φ is evaluated using this value for x. Variable x is used in φ in a constraint of the form $T - x \in I$; this constraint, when evaluated at a point j, checks if $\tau_j - \tau_i \in I$, where τ_j is the time stamp at point j. Here T can be seen as a special variable giving the timestamp of the present

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 23; pp. 23:1–23:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

point. For example, the formula $\varphi = \mathsf{F}x.(a \wedge \mathsf{F}(b \wedge T - x \in [1,2] \wedge \mathsf{F}(c \wedge T - x \in [1,2])))$ asserts that there is a point i in the future where a holds and in its future there is a b within interval $[1,2]$ followed by a c within interval $[1,2]$ from i. In this paper, we restrict ourselves to future time modalities only. Hence, we use the term MTL and TPTL for MTL$[\mathsf{U}_I]$ and TPTL$[\mathsf{U}]$, respectively, and MTL+Past and TPTL+Past for MTL$[\mathsf{U}_I, \mathsf{S}_I]$ and TPTL$[\mathsf{U}, \mathsf{S}]$, respectively. We also confine ourselves to the *pointwise* interpretation of these logics [7].

While these logics are natural formalisms to express real-time properties, it is unfortunate that both the logics have an undecidable satisfiability checking problem, making automated analysis of these logics difficult in general. Exploring natural decidable variants of these logics has been an active area of research since their advent [5, 31, 13, 35, 30, 14, 15]. One of the most celebrated such logics is the *Metric Interval Temporal Logic* (MITL) [1], a subclass of MTL where the timing intervals are restricted to be non-punctual i.e. non-singular (intervals of the form $\langle x, y \rangle$ where $x < y$). The satisfiability checking for MITL formulae is ExpSpace complete [1] (the result also holds for MITL + Past).

Every formula in MTL can be expressed in the 1-variable fragment of TPTL (denoted 1-TPTL). Moreover, the above-mentioned property φ is not expressible in MTL + Past [26]. Hence, 1-TPTL is strictly more expressive than MTL [27, 7]. The Logic 1-TPTL can also express MTL augmented with richer counting and Pnueli modalities. Hence, TPTL is a logic with high expressive power. However, decidable fragments of TPTL are harder to find. While 1-TPTL has decidable satisfiability over finite timed words [10] (albeit with non-primitive recursive complexity), it is undecidable over infinite words [25]. There are no known fragments of multi-variable TPTL which are decidable (without artificially restricting the class of timed words). In this paper, we propose one such logic, which is efficiently decidable over both finite and infinite timed words.

We propose a fragment of TPTL, called $\mathrm{TPTL}_{0,\infty}$, where, for any formula ϕ in negation normal form, each of its closed subformula κ has *unilateral* intervals; that is, intervals of the form $\langle 0, u \rangle$, or of the form $\langle l, \infty)$ (where $\langle \in \{[, (\}$ and $\rangle \in \{],)\}$). The main result of this paper is to show that satisfiability checking for $\mathrm{TPTL}_{0,\infty}$ is PSpace complete. Moreover, we show that even the 1-variable fragment of this logic is strictly more expressive than MITL. PSpace completeness for satisfiability checking is proved as follows: We define a sub-class of Alternating Timed Automata (ATA [24] [21]) called *Very Weak Alternating Timed Automata with Unilateral Intervals*($\mathrm{VWATA}^{0,\infty}$), and show that $\mathrm{VWATA}^{0,\infty}$ have PSpace -complete emptiness checking. A language preserving reduction from $\mathrm{TPTL}_{0,\infty}$ to $\mathrm{VWATA}^{0,\infty}$, similar to [10, 24, 34], completes the proof. To our knowledge, $\mathrm{VWATA}^{0,\infty}$ is amongst the first known fragment of multi-clock alternating timed automata (ATA) with efficiently decidable emptiness checking. Thus, we believe that $\mathrm{TPTL}_{0,\infty}$ and $\mathrm{VWATA}^{0,\infty}$ are interesting novel additions to logics and automata for real-time behaviours.

One of the key challenges in establishing the decidability of $\mathrm{VWATA}^{0,\infty}$ is to show that the configuration sizes can be bounded. In an ATA, a configuration can be unboundedly large owing to several conjunctive transitions, each spawning a state with a new clock valuation. We provide a framework for compressing the configuration sizes of $\mathrm{VWATA}^{0,\infty}$ based on simulation relations amongst states of the $\mathrm{VWATA}^{0,\infty}$. We then prove that such compression yields a simulation-equivalent transition system whose configuration sizes are bounded. This bound allows us to give a subset-like construction resulting in a simulation equivalent (hence, language equivalent) timed automata with polynomially many clocks.

The paper is organized as follows. Section 2 defines the TPTL and ATA, and $(0, \infty)$ fragments of these formalisms. In Section 3, we prove the PSpace emptiness checking of $\mathrm{TPTL}^{0,\infty}$. Section 4 discusses the expressiveness of $\mathrm{TPTL}_{0,\infty}$. Section 5 concludes our work with a discussion on the implication of our work in the field of timed logics and some interesting problems that we leave open.

2 Preliminaries

Let $\mathbb{Z}, \mathbb{Z}_{\geq 0}, \mathbb{N}, \mathbb{R}, \mathbb{R}_{\geq 0}$ respectively denote the set of integers, non-negative integers, natural numbers (excluding 0), real numbers, and non-negative real numbers. Given a sequence $\mathbf{a} = a_1 a_2 \ldots$, $\mathbf{a}[i] = a_i$ denotes the i^{th} element of the sequence, $a[i..j]$ represents $a_i a_{i+1} \ldots a_j$, $a[i..]$ represents $a_i a_{i+1} \ldots$ and $a[..i]$ represents $= a_1 a_2 \ldots a_i$. Let $\mathcal{I}_{\mathsf{int}}$ be the set of all the open, half-open, or closed intervals (i.e. convex subsets of real numbers), such that the endpoints of these intervals are in $\mathbb{N} \cup \{0, \infty\}$. Intervals of the form $[x, x]$ are called punctual; a non-punctual interval is one which is not punctual. For, $\langle \in \{(, [\}$ and $\rangle \in \{],)\}$, an interval of the form $\langle 0, u\rangle$ for $u > 0$ is called *right-sided* while an interval of the form $\langle l, \infty)$ is called *left-sided*. A *unilateral* interval is either left-sided or right-sided. Let $\mathcal{I}^{0}_{\mathsf{int}}, \mathcal{I}^{\infty}_{\mathsf{int}} \subseteq \mathcal{I}_{\mathsf{int}}$ respectively be the set of all *right sided* and *left sided* intervals of the form $\langle 0, u\rangle$, $\langle l, \infty)$, for any $l, u \in \mathbb{Z}_{\geq 0}$. Let $\mathcal{I}^{0,\infty}_{\mathsf{int}} = \mathcal{I}^{0}_{\mathsf{int}} \cup \mathcal{I}^{\infty}_{\mathsf{int}}$. For $\tau \in \mathbb{R}$ and interval $\langle a, b\rangle$, $\tau + \langle a, b\rangle$ stands for the interval $\langle \tau + a, \tau + b\rangle$.

Timed Words. Let Σ be a finite alphabet. A finite (infinite) word over Σ is a finite (infinite) sequence over Σ. The set of all the finite (infinite) words over Σ is denoted by Σ^* (Σ^ω). A finite timed word ρ over Σ is a finite sequence of pairs $(\sigma, \tau) \in (\Sigma \times \mathbb{R}_{\geq 0})^* : \rho = (\sigma_1, \tau_1), \ldots, (\sigma_n, \tau_n)$ where $\tau_i \leq \tau_j$ for all $1 \leq i \leq j \leq n$. Let $dom(\rho) = \{1, 2, \ldots n\}$ be the set of points in ρ. Likewise, an infinite timed word is an infinite sequence $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2) \ldots \in (\Sigma \times \mathbb{R}_{\geq 0})^\omega$, where $\sigma_1 \sigma_2 \ldots \in \Sigma^\omega$, and $\tau_1 \tau_2 \ldots$ is a monotonically increasing infinite sequence of real numbers approaching ∞ (i.e. non-zeno). A finite (infinite) timed language is a set of all finite (infinite) timed words over Σ denoted $T\Sigma^*$ ($T\Sigma^\omega$).

Timed Propositional Temporal Logic (TPTL). The logic TPTL extends LTL with freeze quantifiers and is evaluated on timed words. Formulae of TPTL are built from a finite alphabet Σ using Boolean connectives, as well as the temporal modalities of LTL. In addition, TPTL uses a finite set of real-valued variables called freeze variables or *clocks* $X = \{x_1, \ldots, x_n\}$. Let $\nu : X \to \mathbb{R}_{\geq 0}$ represent a valuation assigning a non-negative real value to each clock. Without loss of generality, we work with TPTL in the negation normal form, where all the negations appear only with atomic formulae. Formulae of TPTL are defined as follows.

$$\varphi ::= a \mid \neg a \mid \top \mid \bot \mid x.\varphi \mid T - x \in I \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \mathsf{U} \varphi \mid \mathsf{G}\varphi$$

where $x \in X$, $a \in \Sigma$, $I \in \mathcal{I}_{\mathsf{int}}$. T denotes the time stamp of the position where the formula is evaluated. The construct $x.\varphi$ is called a *freeze quantifier*, which stores in x, the time stamp of the current position and then evaluates φ. $T - x \in I$ is a constraint on the clock variable x, which checks if the time elapsed since the time x was frozen is in the interval I. Duals of Until; "Unless" and "Release" operators can be expressed using a G and an U operator without compromising on succinctness. **Notice** that, in aid of brevity, we will typically **abbreviate** subformula $T - x \in I$ to $x \in I$. For a timed word $\rho = (\sigma_1, \tau_1) \ldots (\sigma_n, \tau_n)$, $i \in dom(\rho)$ and a TPTL formula φ, we define the satisfiability $\rho, i, \nu \models \varphi$ at a position i of ρ, given a valuation ν of the clock variables.

$$\begin{array}{llll}
\rho, i, \nu & \models & a & \iff & \sigma_i = a, \\
\rho, i, \nu & \models & x.\varphi & \iff & \rho, i, \nu[x \leftarrow \tau_i] \models \varphi, \\
\rho, i, \nu & \models & T - x \in I & \iff & \tau_i - \nu(x) \in I, \\
\rho, i, \nu & \models & \mathsf{G}\varphi & \iff & \forall j > i, \rho, j, \nu \models \varphi, \\
\rho, i, \nu & \models & \varphi_1 \mathsf{U} \varphi_2 & \iff & \exists j > i, \rho, j \models \varphi_2, \text{ and } \forall i < k < j, \rho, k \models \varphi_1.
\end{array}$$

The F and Next operator is defined in terms of U; $\mathsf{F}\phi = \top\mathsf{U}\phi$ and $\mathsf{Next}\phi = \bot\mathsf{U}\phi$. $\mathbf{0}$ denotes a valuation that maps every variable to 0. A TPTL formula φ is said to be closed iff every variable x used in the timing constraint is quantified (or bound) by a freeze quantifier. A formula that is not closed is *open*. Similarly, in any formula φ, a constraint of the form $x \in I$ is open if x is not quantified. For example, $x.y.(a\mathsf{U}(b \wedge x \in (1,2) \wedge y \in (2,3)))$ is a closed formula while $x.(a \wedge \underline{y \in (2,3)})\mathsf{U}y.(b \wedge x \in (1,2))$ is open as the clock y used in the underlined clock constraint is not in the scope of a freeze quantifier for y. Moreover, the underlined constraint $\underline{y \in (2,3))}$ is an open constraint. Notice that open constraints appear only (and necessarily) in open formulae. Satisfaction of closed formulae is independent of the clock valuation; that is, if ψ is a closed formula, then for a timed word ρ and a position i in ρ, either for every valuation ν, $\rho, i, \nu \models \psi$; or for every valuation ν, $\rho, i, \nu \not\models \psi$. Hence, for a closed formula ψ, we drop the valuation ν while evaluating satisfaction, and simply write $\rho, i \ \models \psi$. As an example, the closed formula $\varphi{=}x.(a\mathsf{U}(b\mathsf{U}(c \wedge x{\in}[1,2])))$ is satisfied by the timed word $\rho{=}(a,0)(a,0.2)(b,1.1)(b,1.9)(c,1.91)(c,2.1)$ since $\rho, 1 \models \varphi$. The word $\rho' = (a,0)(a,0.3)(b,1.4)(c,2.1)(c,2.5)$ does not satisfy φ. However, $\rho', 2 \models \varphi$: if we start from the second position of ρ', the value 0.3 is stored in x by the freeze quantifier, and when we reach the position 4 of ρ' with $\tau_4 = 2.1$ we obtain $T - x = 2.1 - 0.3 \in [1,2]$.

Given any closed TPTL formula φ, its language, $L(\varphi) = \{\rho | \rho, 1 \models \varphi\}$, is set of all the timed words satisfying it. We say that a closed formula φ is satisfiable iff $L(\varphi) \neq \emptyset$.

Size of a TPTL formula. Given a TPTL formula φ, the size of φ denoted by $|\varphi|$ is defined as $B{+}M{+}C$ where B is the number of Boolean operators in φ, M is the number of temporal modalities ($\mathsf{G}, \mathsf{U}, \mathsf{Next}, \mathsf{F}$) and freeze quantifiers in φ, and C is obtained by multiplying the number of time constraints in φ with $2{\times}(\lfloor\log(c_{max})\rfloor{+}1)$ where c_{max} is the maximal constant appearing in the time constraints of φ. For example, for $\varphi = x.(a \wedge bU(c \vee x \leq (1,2)))$, $|\varphi| = 2 + 2 + 2 \times (1+1) = 8$ as it contains two boolean operators, one temporal modality, one freeze quantifier and one timing constraint where $c_{max} = 2$.

The subclass of TPTL that uses **only k-clock variables** is known as **k-TPTL**. By [10] [25], satisfiability checking for 1-TPTL is decidable over finite models but non-primitive recursive hard, and undecidable over infinite models. Satisfiability checking for 2-TPTL is undecidable over both finite and infinite models [6] [18]. Towards the main contribution of this paper, we propose a "non-punctual" fragment of TPTL with unilateral intervals, called TPTL$^{0,\infty}$, and show that its satisfiability checking is decidable with multiple variables over both finite and infinite timed words (PSpace -complete). Further, 1-TPTL$^{0,\infty}$ is already more expressive than MITL, which has an ExpSpace -complete satisfiability checking.

2.1 Multi-clock TPTL with unilateral intervals: TPTL$^{0,\infty}$

We say that a formula φ is of the type $\leq$ ($\geq$), iff all the intervals appearing in the open constraints of φ are in $\mathcal{I}^0_{\mathsf{int}}$ ($\mathcal{I}^\infty_{\mathsf{int}}$). Notice that a closed formula belongs to both types $\leq$ and $\geq$. There are open formulae that are neither of type $\leq$ nor $\geq$. A TPTL formula φ in negation normal form is a TPTL$^{0,\infty}$ formula iff every subformula of φ is either of the type $\leq$ or $\geq$. For example, $x.y.(a\mathsf{U}(b\mathsf{U}(c \wedge x < 3 \wedge y \leq 2 \wedge x.(\mathsf{Next}(c \wedge x > 1)))))$ is a TPTL$^{0,\infty}$ formula since there is no subformula that doesn't belong to either types $\leq$ or $\geq$. However, $x.y.(a\mathsf{U}(b \wedge x \leq 3 \wedge y \geq 5))$ is not TPTL$^{0,\infty}$, since $(b \wedge x \leq 3 \wedge y \geq 5)$ is of neither type $\leq$ or $\geq$ as the open constraints within this subformula use both left-sided as well as right-sided intervals. This restriction is inspired by that of MITL$^{0,\infty}$. Any MITL$^{0,\infty}$ formula can be expressed in 1-TPTL$^{0,\infty}$ by applying the same reduction from MITL to 1-TPTL (see Remark 15). Next, we introduce alternating timed automata which are useful in proving the main result, i.e., Theorem 2.

2.1.1 Alternating Timed Automata

An Alternating Timed Automata (ATA) is a 7-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G})$, where, Q is a finite set of locations, X is a finite set of clock variables, $\mathcal{G}$ is a finite set of guards of the form $x \in I$ where $I \in \mathcal{I}_{\mathsf{int}}$ and $x \in X$, δ is a transition function, $q_0 \in Q$ is the initial location, and $\mathsf{Q}_{\mathsf{acc}} \subseteq Q$ is a set of accepting locations. The transition function is defined as $\delta : Q \times \Sigma \mapsto \Phi(Q, \mathcal{G})$ where $\Phi(Q, \mathcal{G})$ is defined by the grammar $\varphi ::= \top|\bot|\varphi_1 \wedge \varphi_2|\varphi_1 \vee \varphi_2|q|x \in I|Y.q$ with $q \in Q$, $x \in X$, $(x \in I)$ is a guard in $\mathcal{G}$, $Y \subseteq X$, Y is not the empty set. $\top, \bot$ respectively denote True and False. $Y.q$ is a *binding construct* which resets all clocks in Y to zero after taking the transition. Let $p, q \in Q$ and $Y \subseteq X$. We say that there is a transition from p to q iff q appears in $\delta(p, b)$ for some $b \in \Sigma$. We say that there is a **strong reset transition**, **non-reset** transition, and a **Y-reset** transition from location p to q iff for some $b \in \Sigma$, $X.q$, q, and $Y.q$, respectively, appears in $\delta(p, b)$ for some $b \in \Sigma$. The 1-clock restriction of ATA has been considered in [24] and [21].

Evaluation of $\Phi(Q, \mathcal{G})$. Given an ATA $\mathcal{A}$, a state s is defined as a pair consisting of a location and a valuation over X, i.e., $s \in Q \times \mathcal{V}_X$. A configuration C of an ATA is a finite set of states. Let S and $\mathcal{C}$ respectively denote the set of all states and configurations of $\mathbb{A}$. A configuration C and a clock valuation ν define a Boolean valuation for $\Phi(Q, \mathcal{G})$ as follows:

$$\begin{array}{l|l} C \models_\nu q \text{ iff } (q, \nu) \in C, & C \models_\nu Y.q \text{ iff } (q, \nu) \in C, \text{ and } \forall x \in Y. \nu(x) = 0, \\ C \models_\nu x \in I \text{ iff } \nu(x) \in I, & C \models_\nu \varphi_1 \wedge \varphi_2 \text{ iff } C \models_\nu \varphi_1 \wedge C \models_\nu \varphi_2, \\ C \models_\nu \top \text{ for all } C \in \mathcal{C}, & C \models_\nu \varphi_1 \vee \varphi_2 \text{ iff } C \models_\nu \varphi_1 \vee C \models_\nu \varphi_2. \end{array}$$

Finally, $C \not\models_\nu \bot$ for all possible configurations. We say that C is a minimal model for $\varphi \in \Phi(Q, \mathcal{G})$ with respect to ν (denoted by $C \models_\nu^{\mathsf{min}} \varphi$) iff $C \models_\nu \varphi$ and no proper subset C' of C is such that $C' \models_\nu \varphi$. See Figure 1 in the full version for the graphical representation of the ATA.

Semantics of ATA. Given a state $s = (q, \nu)$, a time delay $t \in \mathbb{R}_{\geq 0}$ and $a \in \Sigma$, the successors of $s = (q, \nu)$ on time delay t followed by a is any configuration C such that $C \models_{\nu+t}^{\mathsf{min}} \delta(q, a)$. $\mathsf{Succ}_{\mathcal{A}}^{st}(s, t, a)$ is the set of all such successors. The notion of a successor is extended to a configuration in a straightforward manner. A configuration C' is a successor of configuration $C = \{s_1, s_2, \ldots s_k\}$ on time delay t and $a \in \Sigma$ (denoted by $C \xrightarrow{(t,a)}_{\mathcal{A}} C'$) iff $C' = C_1 \cup \ldots \cup C_k$ such that $\forall 1 \leq i \leq k, C_i \in \mathsf{Succ}_{\mathcal{A}}^{st}(s_i, t, a)$. We denote by $\mathsf{Succ}_\delta(C, t, a)$ set of all such successors C'.

The initial configuration is defined by $C_{init} = \{(q_0, \mathbf{0})\}$, and a configuration C is accepting iff for all $s \in C$, s is an accepting state, that is $s = (q, \nu)$ for $q \in \mathsf{Q}_{\mathsf{acc}}$. Let $\mathcal{C}_{acc}$ be the set of all the accepting configurations. Hence, the empty configuration is an accepting configuration. We define the semantics of ATA using a Labelled Transition System (LTS). An LTS is a 5-tuple $T = (S, s_0, \Sigma, \delta, S_f)$, where S is a finite or infinite set of states, $s_0 \in S$ is the initial state, Σ is set of symbols, $\delta : S \times \Sigma \times S$ is a transition relation, and $S_f \subseteq S$ is a set of final states. A (finite) run $\mathbf{R}$ of an LTS is a (finite) sequence of the form $s_0, a_1, s_1, a_2, s_2, a_3 \ldots$ where $s_1, s_2, \ldots \in S$ are states of T, and $a_1, a_2, \ldots$ are symbols in Σ such that for all $i > 0$, $s_i \in \delta(s_{i-1}, a_i)$. We say that a run $R = s_0, a_1, s_1, a_2, s_2, a_3 \ldots$ visits a state s (or visits a set of states S') iff the sequence R contains s (or contains states in S'). A run is said to be accepting iff it ends in some state $s \in S_f$. Similarly, an infinite run is said to be Büchi accepting iff it visits S_f infinitely often.

Runs of $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G})$ starting from a configuration C are the runs of LTS $TS(\mathcal{A}, C) = (\mathcal{C}, C, \mathbb{R}_{\geq 0} \times \Sigma, \rightarrow, \mathcal{C}_{acc}$). Notice that the states of LTS $TS(\mathcal{A}, C)$ are configurations of $\mathcal{A}$ (i.e., a set of states of $\mathcal{A}$ and not just the states of $\mathcal{A}$). Let $\rho = (a_1, \tau_1)(a_2, \tau_2)\ldots$ be any timed word over Σ. We say that a run $R = C, (t_1, a_1), C_1, (t_2, a_2)\ldots$ is produced by $\mathcal{A}$ on ρ starting from a configuration C iff $C \xrightarrow{(t_1,a_1)} C_1 \xrightarrow{(t_2,a_2)} C_2 \ldots$ where $t_i = \tau_i - \tau_{i-1}$ for $i > 0$ and $\tau_0 = 0$. Let $\mathcal{A}(\rho, C)$ be the set of all the runs produced by $\mathcal{A}$ on ρ, starting from the configuration C. We denote $TS(\mathcal{A}, C_{init})$ as simply $TS(\mathcal{A})$. A run starting from the initial configuration C_{init} is called an initialized run. We denote $\mathcal{A}(\rho, C_{init})$ by $\mathcal{A}(\rho)$. ρ, i is said to be accepted (Büchi accepted) by $\mathcal{A}$ starting with configuration C, denoted by $\rho, i \models \mathcal{A}, C$, iff there exists a run in $\mathcal{A}(\rho[i..], C)$ accepted (Büchi accepted) by $TS(\mathcal{A})$ (i.e., simulating $\mathcal{A}$ on the suffix of ρ starting at position i we obtain an accepting run). We say that ρ is accepted by $\mathcal{A}$ iff $\rho, 1 \models C_{init}$.

We define the finite (infinite) language of $\mathcal{A}$, denoted by $L_{fin}(\mathcal{A})$ ($L_{inf}(\mathcal{A})$), as a set of all the finite (infinite) timed words accepted by $\mathcal{A}$. When clear from context, we drop the subscript in L_{fin} and L_{inf}.

Non-Deterministic Timed Automata (NTA) is a subclass of ATA where $\Phi(Q, \mathcal{G})$ is restricted to be in disjunctive normal form (DNF), where each disjunct is of the form $(q \wedge x \in I)$ or $(X'.q \wedge x \in I)$. Hence, for any $s \in S$, $t \geq 0$, $a \in \Sigma$ and any configuration $C \in \mathsf{Succ}^{st}_{\delta}(s, t, a)$ implies $C \leq 1$.

We call the ATA $\mathcal{A}$ a **Very Weak ATA** (VWATA) iff (1) there is a partial order $\ll_{\mathcal{A}} \subseteq Q \times Q$ such that there is a transition from p to q iff $q \ll_{\mathcal{A}} p$, (2) all the self-loop transitions (transitions entering and exiting into the same location) are non-reset transitions, and (3) For every location q, there is at most one location $p \neq q$ such that there is a transition from p to q. Moreover, all the transitions from p to q reset the same set of clocks. This makes the transition diagram of VWATA a tree and not a DAG (excluding self-loops).

▶ Remark 1. In the literature, VWATA (also called Partially-Ordered Alternating Timed Automata in [20]) and their corresponding untimed version [9, 32](also called as Linear [22], Linear-Weak [11], 1-Weak [28], and Self-Loop [33] Alternating Automata) are required to satisfy only conditions (1) and (2). It can be shown that condition (3) does not affect the expressiveness of the machine. We notice that this version of VWATA is enough to express TPTL formulae efficiently (linear in the size of TPTL formulae). In case of translation from TPTL to VWATA satisfying condition (3) the number of locations in the resulting ATA will depend on the size of the formula tree. On the other hand, the total number of locations depends on the formula DAG on similar translation from TPTL to VWATA satisfying only (1) and (2) making it exponentially more succinct. Hence, we consider a less succinct representation (i.e., tree or string, which is standard) of TPTL formulae for computing its size as compared to the DAG representation.

2.1.2 ATA with Unilateral Intervals: ATA$^{0,\infty}$

Similar to the unilateral version of TPTL (i.e. TPTL$^{0,\infty}$), we define a unilateral version of ATA, i.e., ATA$^{0,\infty}$ as follows. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G})$ be any ATA. Let $\mathcal{G}^{\geq}$ ($\mathcal{G}^{\leq}$) be the subset of $\mathcal{G}$ containing all the guards of the form $x \in I$ where $I \in \mathcal{I}^{\infty}_{\mathsf{int}}$ ($I \in \mathcal{I}^{0}_{\mathsf{int}}$). $\mathcal{A}$ is said to be an ATA$^{0,\infty}$ iff, Q can be partitioned into $\mathsf{Q}^{\geq}$ and $\mathsf{Q}^{\leq}$ any transition exiting from any location $q \in \mathsf{Q}^{\geq}$ ($q \in \mathsf{Q}^{\leq}$) is guarded by a guard in $\mathcal{G}^{\geq}$ ($\mathcal{G}^{\leq}$), and any transition from any location in $\mathsf{Q}^{\geq}$ to a location in $\mathsf{Q}^{\leq}$, or vice-versa, is a strong reset transition. $\mathcal{A}$ is said to be VWATA$^{0,\infty}$ iff it is an ATA$^{0,\infty}$, and a VWATA. From this point onwards, for any set of locations Q of ATA$^{0,\infty}$, $\mathsf{Q}^{\geq}$ and $\mathsf{Q}^{\leq}$ will denote partitions of Q satisfying the above condition.

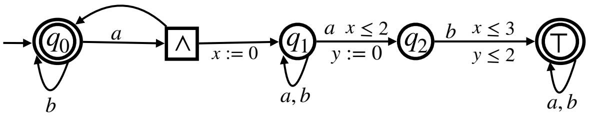


Figure 1 VWATA$^{0,\infty}$ equivalent to φ. Location q_i corresponds to the subformula φ_i: $\rho, i, \nu \models \varphi_i$ iff $\rho, i \models (q_i, \nu)$.

3 Satisfiability Checking for TPTL$^{0,\infty}$

This section is dedicated to proving the following main theorem of this paper.

▶ **Theorem 2.** *Satisfiability Checking for TPTL$^{0,\infty}$ is PSPACE -Complete*

PSPACE hardness follows from the hardness of satisfiability checking of the sublogics LTL and MITL$^{0,\infty}$ (see section 4.1 for the details on MITL$^{0,\infty}$). To show membership in PSPACE we propose the following steps: (1) We reduce any given k-TPTL$^{0,\infty}$ formula φ, to an equivalent VWATA$^{0,\infty}$, $\mathcal{A}$, with k clock variables and at most $|\varphi| + 1$ number of locations. (2) We give a novel on-the-fly construction from any VWATA$^{0,\infty}$ to simulation equivalent NTA $\mathbb{A}$ with exponential blow-up in the number of locations and polynomial blow-up in the number of clocks. Hence, the region automata corresponding to $\mathbb{A}$ has at most exponentially many states, and thus each state can be represented in polynomial space. [1]

▶ Remark 3. Notice that while the reduction from VWATA$^{0,\infty}$ to timed automata results in an exponential blow-up in the number of locations we can directly construct the region automaton of the corresponding timed automaton on-the-fly making sure that we need at most polynomial space to solve its emptiness checking problem.

We demonstrate our steps of construction using a running example. For the formal constructions please refer to the full version. In our running example, we start with the given formula $\varphi = \mathsf{G}(\neg a \vee x.(\mathsf{F}(a \wedge T - x \leq 2 \wedge y.\mathsf{Next}(b \wedge T - x \leq 3 \wedge T - y \leq 2)))$.

3.1 TPTL$^{0,\infty}$ to VWATA$^{0,\infty}$

This step is a straightforward multi-clock generalization of translation from MTL and 1-TPTL to 1-ATA in [24] and [10], respectively, (which are themselves timed generalization of reduction from LTL to Very Weak Alternating Automata [34] [9]). We give the reduction in the full version for completeness. The proof of equivalence is identical to that in [24] and [10] resulting in the following Theorem 4. We give the VWATA$^{0,\infty}$ corresponding to the formula φ of the running example in Figure 1. Hence, to prove the main theorem it suffices to show that emptiness checking for VWATA$^{0,\infty}$ is in PSPACE (i.e. Theorem 5).

▶ **Theorem 4.** *Any k variable TPTL formula φ over Σ can be reduced to an equivalent VWATA, $\mathcal{A} = (Q, 2^{\Sigma}, \delta, init, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G})$, with $|X| = k$, $|Q| \leq |\varphi| + 1$, and $\mathcal{G}$ is the set of all the guards appearing in φ. Moreover, if φ is a TPTL$^{0,\infty}$ formula, then the $\mathcal{A}$ is VWATA$^{0,\infty}$.*

[1] While one can argue about existence of a simple reduction from TPTL$^{0,\infty}$ to Recursive Memory Event Clock Logic of [17] using projections, we would still need to show that such a reduction requires only bounded memory which can be non-trivial, especially with multiple clocks. We believe that the automata-theoretic argument in this paper is a clean technique for proving such bounds.

3.2 Emptiness Checking for VWATA$^{0,\infty}$

The following theorem is the main technical result.

▶ **Theorem 5.** *Emptiness Checking for VWATA$^{0,\infty}$ is in PSPACE .*

We give a translation from VWATA$^{0,\infty}$ $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G})$ to an equivalent timed automaton, $\mathbb{A} = (\mathcal{Q}, \Sigma, \Delta, \mathsf{q}_0, \mathcal{Q}_{acc}, \mathcal{X}, \mathcal{G})$, such that the transition system of $\mathcal{A}$ (i.e., $TS(\mathcal{A})$) is simulation equivalent to that of $\mathbb{A}$ (i.e., $TS(\mathbb{A})$). Hence, by the Proposition 6, $L(\mathcal{A}) = L(\mathbb{A})$.

Moreover, $\mathcal{Q} = O(2^{Poly(Q)})$ and $|\mathcal{X}| = |X| \times |Q|$. Hence, the number of states in the corresponding region automaton is exponential to the size of $\mathcal{A}$ (i.e. $O(2^{Poly(|Q|,|X|)}) \times (2 \times c_{max} + 1)$ where c_{max} is the maximum constant used in the constraints appearing in $\mathcal{G}$). Hence, each state of the region automata (when encoded in binary) can be represented in polynomial space proving membership in PSPACE . We prove the above by giving a translation from VWATA$^{0,\infty}$ to timed automata with polynomial blowup in the number of clocks and exponential blowup in the set of locations. As a side-effect, we also show that emptiness checking for 1-ATA$^{0,\infty}$ is in PSPACE (using the same construction) generalizing the result of [16]. We first briefly discuss the concept of simulation relations and preorders.

3.2.1 Simulation Relations and Preorder

We fix a pair of labeled transition system, $TS^1{=}(S^1, s_0^1, \Sigma, \delta^1, S_f^1)$ and $TS^2{=}(S^2, s_0^2, \Sigma, \delta^2, S_f^2)$. A relation $\preceq\, \subseteq S^1 \times S^2$ is a simulation relation iff (1) $s_0^1 {\preceq} s_0^2$, (2) for every $s_1 \preceq s_2$, (2.1) if $s_1 \in S_f^1$ then $s_2 \in S_f^2$, and (2.2) for every $a \in \Sigma$, for every $s_1' \in \delta(s_1, a)$ there exists $s_2' \in \delta(s_2, a)$ such that $s_1' \preceq s_2'$. If $s_1 \preceq s_2$, then we say that s_2 simulates s_1 wrt $\preceq$.

Let $S = S^1 \cup S^2$. Notice that simulation relations are closed under union. Hence, there is a unique maximal simulation relation, $\leq\, \subseteq S \times S$, which is the union of all the simulation relations amongst states of TS^1 and TS^2 (i.e. all the simulation relations between TS^1 and itself, between TS^2 and itself, and from TS^1 to TS^2 and vice-versa). Notice that $\leq$ is a preorder relation (i.e. reflexive and transitive), and hence also called simulation preorder. Similarly, simulation equivalence relation, $\cong$ is defined as the largest symmetric subset of simulation preorder, $\leq$. I.e., $s \cong s'$ iff $s \leq s'$ and $s' \leq s$. Hence, it is clear that $\cong$ is an equivalence relation. If $s \leq s'$ we say that s' simulates s. Recall that the states of $TS(\mathcal{A}, C)$ for any ATA $\mathcal{A}$ and its configuration C are configurations of $\mathcal{A}$. Then,

▶ **Proposition 6.** *Let $\mathcal{A}$ and $\mathcal{A}'$ be any ATA, and s_0, s_0' be their initial states, respectively. $TS(\mathcal{A}, \{s\}) \leq TS(\mathcal{A}', \{s'\})$ implies $L_{fin}(\mathcal{A}) \subseteq L_{fin}(\mathcal{A}')$ and $L_{inf}(\mathcal{A}) \subseteq L_{inf}(\mathcal{A}')$. Hence, $TS(\mathcal{A}, \{s\}) \cong TS(\mathcal{A}', \{s'\})$ implies $L_{fin}(\mathcal{A}) = L_{fin}(\mathcal{A}')$ and $L_{inf}(\mathcal{A}) = L_{inf}(\mathcal{A}')$*

We fix an ATA $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G})$. Let C and C' be arbitrary configurations of $\mathcal{A}$. Let $\leq_{\mathcal{A}}$, $\cong_{\mathcal{A}}$ be the simulation preorder and simulation equivalence amongst configurations of $\mathcal{A}$. That is, $C \leq_{\mathcal{A}} C'$ iff C' simulates C, and $C \cong_{\mathcal{A}} C'$ iff C is simulation equivalent to C' in $TS(A)$, the transition system corresponding to ATA $\mathcal{A}$. Then, by Proposition 6:

▶ Remark 7. For any configuration C and C' of A, $C \leq_{\mathcal{A}} C'$ implies $L(\mathcal{A}, C) \subseteq L(\mathcal{A}, C')$ and $C \cong_{\mathcal{A}} C'$ implies $L(\mathcal{A}, C) = L(\mathcal{A}, C')$.

▶ Remark 8. $C \supseteq C'$ implies $C \leq_{\mathcal{A}} C'$. Hence, for any timed word ρ, if $\rho, i \models \mathcal{A}, C$ then $\rho, i \models \mathcal{A}, C'$. Intuitively, the additional states in C (which are not appearing in C') impose extra obligations in addition to that imposed by states common in both C and C' which makes reaching the accepting configuration (hence accepting a timed word) harder from C. For formal proof, please refer to the full version.

▶ Remark 9. If $D' \subseteq C$ and $D \leq_{\mathcal{A}} D'$, then $(C \setminus D') \cup D \leq_{\mathcal{A}} C$. In other words, we can replace the states in D' with that in D in any configuration C, and get a configuration that is simulated by C. Hence, $L(\mathcal{A}, (C \setminus D') \cup D) \subseteq L(\mathcal{A}, C)$.

Proof outline of Remark 9. First, show that for any configurations E_1, E_2, and E if both $E_1$and E_2 individually simulate E, then $(E_1 \cup E_2)$ simulates E. Second, substitute $E_1 = C \setminus D', E_2 = D'$, and $E = (C \setminus D') \cup D$. By Remark 8, E_1 and D individually simulate E. $E_2 = D'$ simulates D is given. Hence, E_2 simulates E by transitivity of preorders. Thus, $(E_1 \cup E_2)$ simulates E proving our remark. For full proof please refer to the full version. ◀

Both the above remarks imply the following Proposition. We abuse the notation by writing $\{s\} \leq_{\mathcal{A}} \{s'\}$ as $s \leq_{\mathcal{A}} s'$.

▶ **Proposition 10.** *If $s, s' \in C$ and $s \leq_{\mathcal{A}} s'$ then $C \setminus \{s'\} \cong_{\mathcal{A}} C$.*

Proof. Notice that $(C \setminus \{s'\}) \cup \{s\} = C \setminus \{s'\}$. Hence, by Remark 9, $(C \setminus \{s'\}) \leq_{\mathcal{A}} C$. By Remark 8, $C \leq_{\mathcal{A}} C \setminus \{s'\}$. Hence proved. ◀

We use the above Proposition 10 and Lemma 14 (which holds for $\text{VWATA}^{0,\infty}$ and $\text{1-ATA}^{0,\infty}$) to bound the cardinality of the configuration preserving simulation equivalence. This bound on the cardinality of configurations will imply that we need to remember only a bounded number of clock values to simulate these configurations. Hence, we use this bound on the cardinality of the configurations to bound the number of clock copies required while constructing the required timed automaton.

3.2.2 Bounding Cardinality of Configurations

Intuition

We now discuss the intuition for the decidability of $\text{VWATA}^{0,\infty}$. The main reason for the undecidability of ATA or VWATA is due to the unboundedness of the configuration size. That is, the cardinality of the configurations could depend on the length of the timed word prefix read so far. Hence, we need to keep track of an unbounded number of clocks. This happens, because we can reset a clock x in one branch and not reset x in another branch while taking transitions. This is a result of transitions containing clauses of the form $(X_i.q_i \wedge X_j.q_j)$ where $X_i \neq X_j$ and $X_i, X_j \subseteq X$. That is, we get two states in the successive configuration each resetting a different set of clocks. Hence, we need to remember multiple values for clock variables that are reset in one branch and not in another. In case of $\text{ATA}^{0,\infty}$, we observe the following:

- Observation 1 – Let $q \in \mathsf{Q}^{\geq}$. Due to the nature of constraints, i.e. $x_i \in (l, \infty)$, if we have a pair of states $(q, \nu_1), (q, \nu_2)$ in a configuration C, such that $\nu_1 \leq \nu_2$ (i.e. $\forall x \in X. \nu_1(x) \leq \nu_2(x)$), then any timing constraint that is satisfied by ν_1 will also be satisfied by ν_2. Hence, any transition that can be taken by (q, ν_1) can also be taken by (q, ν_2). Moreover, after taking the same transition (time delay followed by event-based transition) both (q, ν_1) and (q, ν_2) get states of the form (q', ν_1') and (q', ν_2'), respectively, in their successor configurations, such that $\nu_1' \leq \nu_2'$ if $q' \in \mathsf{Q}^{\geq}$ and $\nu_1' = \nu_2' = \mathbf{0}$ if $q' \in \mathsf{Q}^{\leq}$ Hence, by Proposition 10, we can delete (q, ν_2) from C preserving simulation equivalence (and hence the language). A similar argument applies for $q \in \mathsf{Q}^{\leq}$.
- Observation 2 – In 1-ATA, for any pair of valuations ν_1, ν_2, either $\nu_1 \leq \nu_2$ or $\nu_2 \leq \nu_1$. Hence, on applying the reduction using Proposition 10 (and discussed in the previous bullet, i.e., Observation 1), we will always get a configuration, where each location appears at most once. Hence, the configuration size is bounded by the number of locations.

- Observation 3 – But this is not necessarily the case for multiple clocks. This is because there could be unboundedly many incomparable valuations. For example, for 2-clocks $X = \{x, y\}$, consider the following family of configurations parameterized by m, $C_m = \{(q, x = 0.1 + nk, y = 0.9 - nk) | n \in \{0, \dots, m-1\}\}$ and $k = 0.8/m$. $|C_m| = m$ and all the clock valuations are incomparable. Notice
$C_8 = \{(q, x = 0.1, y = 0.9), (q, x = 0.2, y = 0.8) \dots (q, x = 0.9, y = 0.1)\}$.
Hence, as the second main step we show that, if $\mathcal{A}$ is a VWATA$^{0,\infty}$, and if we conservatively keep on compressing the configurations as discussed in Observation 1 (using Proposition 10), we will have boundedly many incomparable clock valuations. To be precise, we will have at most one copy of each location in the configuration. This is shown in Lemma 14.

Bounding Lemma

In this section, we will use the intuition in Observation 1 for constructing a simulation equivalent transition system for a given 1-ATA$^{0,\infty}$ and VWATA$^{0,\infty}$ whose states are configurations of given ATA $\mathcal{A}$ with bounded cardinality. For the 1-ATA$^{0,\infty}$, the intuition in Observation 2 guarantees the case. For the multi-clock VWATA$^{0,\infty}$, the issues discussed in Observation 3 must be resolved. This is resolved in Lemma 14, the main contribution of this section. In what follows, assume $\mathcal{A}$ to be an ATA$^{0,\infty}$. We define relation $\preceq$ amongst states of $\mathcal{A}$. For $\sim \in \{\leq, \geq\}$, let $\preceq$ be defined between states such that $s \preceq s'$ iff $s = (q, \nu)$, $s' = (q, \nu')$, and if $q \in \mathsf{Q}^{\sim}$ then $\nu' \sim \nu$. By Observation 1 we have Proposition 11. The formal proof appears in the full version.

▶ **Proposition 11.** *$s \preceq s'$ implies $s \leq_{\mathcal{A}} s'$.*

Given any configuration C, we define $\mathsf{Red}_{\preceq}(C)$ as a configuration C' obtained from C, by deleting all states $s' \in C'$ if there exists a state $s \in C'$, such that $s \neq s'$, and $s \preceq s'$. Intuitively, we delete some information from a configuration that is redundant in deciding whether a timed behaviour from that state is accepted or not.

Let C_0 be the initial configuration of $\mathcal{A}$. Let $TS(\mathcal{A}) = (\mathcal{C}, C_0, (\mathbb{R}_{\geq 0} \times \Sigma), \rightarrow_{\mathcal{A}})$ be the transition system corresponding to $\mathcal{A}$. We define $\mathsf{T}_{red}(\mathcal{A})$ as a transition system $\mathsf{T}_{red}(\mathcal{A}) = (\mathcal{C}, C'_0, (\mathbb{R}_{\geq 0} \times \Sigma), \rightarrow_{\mathcal{A},\mathsf{red}})$ such that $C'_0 = \mathsf{Red}_{\preceq}(C_0)$ and for any $C, C', D, D' \in \mathcal{C}$, $a \in \Sigma$, and $t \in \mathbb{R}_{\geq 0}$, $C \xrightarrow{(t,a)}_{\mathcal{A}} C'$ iff $D \xrightarrow{(t,a)}_{\mathcal{A},\mathsf{red}} D'$, $D = \mathsf{Red}_{\preceq}(C)$, and $D' = \mathsf{Red}_{\preceq}(C')$. By Proposition 12, $TS(\mathcal{A})$ is simulation equivalent to $\mathsf{T}_{red}(\mathcal{A})$. The following Proposition is implied by Proposition 10 and 11.

▶ **Proposition 12.** *$C \cong_{\mathcal{A}} \mathsf{Red}_{\preceq}(C)$. Hence, $TS(\mathcal{A})$ and $\mathsf{T}_{red}(\mathcal{A})$ are simulation equivalent.*

▶ Remark 13. Any run R' is a run of $\mathsf{T}_{red}(\mathcal{A})$ iff $R' = \mathsf{Img}(R)$ for some run R of $\mathcal{A}$, where $\mathsf{Img}(R)$ is defined as follows. $R = C_0 \xrightarrow{(t_0,a_0)}_{\mathcal{A}} C_1 \xrightarrow{(t_1,a_1)}_{\mathcal{A}} C_2 \dots$, we define $\mathsf{Img}(R)$ as run $R' = C''_0 \xrightarrow{(t_0,a_0)} C''_1 \xrightarrow{(t_1,a_1)} C''_2 \dots$ where $C'_0 = C''_0 = \mathsf{Red}_{\preceq}(C_0)$ and $\forall i \geq 0. C''_i \xrightarrow{(t_i,a_i)}_{\mathcal{A}} C'_i$ and $C''_i = \mathsf{Red}_{\preceq} C'_i$.

▶ **Lemma 14.** *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G})$ be either an 1-ATA$^{0,\infty}$ or VWATA$^{0,\infty}$. Let R be a run of $\mathcal{A}$, and $R' = \mathsf{Img}(R) = C''_0(t_0, a_0)C''_1(t_1, a_1) \dots$, then for all $i \geq 1$, C''_i does not contain states (q, ν) and (q, ν') where $\nu \neq \nu'$ for any $q \in Q$. In other words, every location $q \in Q$ appears at most once in any configuration C''_i for any $i \geq 1$. Hence, $|C''_i| \leq |Q|$.*

Proof (sketch). Notice that if $\mathcal{A}$ was 1-ATA$^{0,\infty}$, the above statement is straightforward as no two clock valuations are incomparable in the case of 1-clock. We now show the same for $\mathcal{A}$ being a multi-clock VWATA$^{0,\infty}$. We just present intuition behind the proof idea. A

formal proof is proved using DAG semantics of ATA and can be found in the full version. We prove this by contradiction. Assumption 1 - Suppose k is the smallest number such that C''_k contains two copies of some location $q \in Q$. Hence, there exists ν and ν' such that ν' is incomparable to ν and $(q, \nu), (q, \nu') \in C''_k$. Then, the following cases are possible:
Case 1 - Both $(q, \nu), (q, \nu')$ appeared from the same location p in C''_{k-1}. But, by condition (3) of VWATA, all the transitions from location p to location q reset the same set of clocks. Moreover, by assumption 1, location p appears at most once in C''_{k-1}. Let $(p, \nu_p) \in C''_{k-1}$. Then both the clock valuations ν and ν' should be identical as they result from the same state (p, ν_p) resetting the same set of clocks.
Case 2 - $(q, \nu), (q, \nu')$ appeared from distinct location (p, ν_{k-1}) and (p', ν'_{k-1}) in C''_{k-1}. By condition (3) of VWATA there is at most one location $q' \neq q$ from which there are transitions entering location q. Moreover, all these transitions reset the same set of clocks. Hence, one of p and p' has to be q. Wlog $p = q$. It suffices to show that whenever such a case occurs, the clock valuation of the state that results from the self-loop (in this case ν) is always greater than or equal to the valuation from the other (in this case ν') (Statement 1). Hence, $\nu' \leq \nu$ which leads to a contradiction. We just present the intuition with an example. Let $\rho = (a_1, \tau_1), (a_2, \tau_2)$. Suppose, $(q_0, \mathbf{0})$ is the initial location of the automaton as drawn in Figure 2. Let $k = 2$. Notice the run in the Figure, $C_1 = \{(q_0, \nu_1), (q_1, \nu'_1)\}$ where if $x \in X'$, $\nu'_1(x) = 0 \leq \nu_1(x) = \tau_1$. Else, $\nu_1(x) = \nu'_1(x) = \tau_1$. Similarly, $C_2 = \{(q_0, \nu_2), (q_1, \nu'), (q_1, \nu)\}$, where (q_1, ν) results from the self loop and (q_1, ν') results from the transition from q_0. Hence, if $x = X'$, $\nu(x) = 0 \leq \nu'(x) = \tau_2 - \tau_1$. Else, $\nu_1(x) = \nu'_1(x) = \tau_2$. In other words, while reaching both (q_1, ν) and (q_1, ν') from the initial configuration, the same set of clock X' was reset. But, in the case of the former, they were reset before the latter. Hence, ν and ν' agree on all the clock values not in X' and $\nu \geq \nu'$ for all the clocks in X. Applying this argument inductively we can prove Statement 1. We believe it is more intuitive to prove the result using the DAG semantics of ATA. Hence, the full proof can be found in the full version, where we introduce the semantics too. ◀

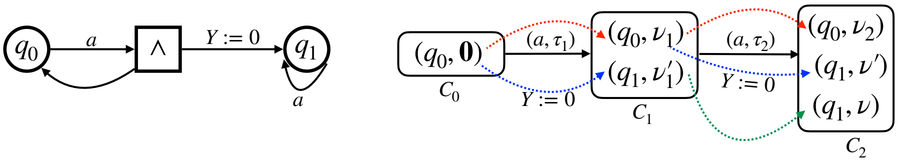


Figure 2 The red and green transitions denote those without resets, and the blue ones with resets. Notice the paths from C_0 to C_2. The Blue-Green and Red-Blue path reset the same set of clocks Y. But the former resets the clocks earlier (in the first step) as compared to the latter (in the second step). Hence in the former, clocks in Y get a chance to progress between C_1 and C_2. Moreover, both the paths should agree on the value of clocks not in Y as they are not reset in both these paths. Hence, $\nu' \leq \nu$.

3.2.3 From VWATA$^{0,\infty}$ to Timed Automata

In this section, we propose an on-the-fly construction from VWATA$^{0,\infty}$ to Timed Automata. The termination relies on Lemma 14. The main idea is to bind the number of active clocks using Lemma 14. Given a VWATA$^{0,\infty}$ or $1-$ATA$^{0,\infty}$, $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathsf{Q}_{\mathsf{acc}}, X, \mathcal{G}, \mathsf{Q}^{\geq}, \mathsf{Q}^{\leq})$ we get a timed automaton $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, q'_0, \mathcal{Q}_{acc}, X \times \{0, \ldots, |Q| - 1\}, \mathcal{G})$ and at every step we reduce the size of the location $q \in \mathcal{Q}$ preserving simulation equivalence. Let V be set of

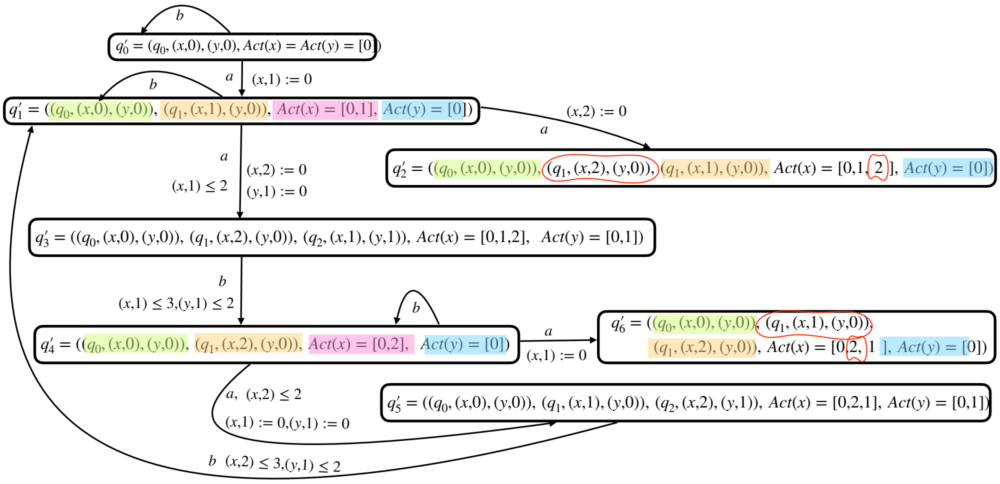

b
q'_0 = (q_0, (x,0), (y,0), Act(x) = Act(y) = [0])
a (x,1) := 0
b
q'_1 = ((q_0, (x,0), (y,0)), (q_1, (x,1), (y,0)), Act(x) = [0,1], Act(y) = [0])
(x,2) := 0
a
q'_2 = ((q_0, (x,0), (y,0)), (q_1, (x,2), (y,0)), (q_1, (x,1), (y,0)), Act(x) = [0,1, 2], Act(y) = [0])
a
(x,1) ≤ 2
(x,2) := 0
(y,1) := 0
q'_3 = ((q_0, (x,0), (y,0)), (q_1, (x,2), (y,0)), (q_2, (x,1), (y,1)), Act(x) = [0,1,2], Act(y) = [0,1])
b
(x,1) ≤ 3,(y,1) ≤ 2
b
q'_4 = ((q_0, (x,0), (y,0)), (q_1, (x,2), (y,0)), Act(x) = [0,2], Act(y) = [0])
a
(x,1) := 0
q'_6 = ((q_0, (x,0), (y,0)), (q_1, (x,1), (y,0)), (q_1, (x,2), (y,0)), Act(x) = [0,2, 1], Act(y) = [0])
a, (x,2) ≤ 2
(x,1) := 0,(y,1) := 0
q'_5 = ((q_0, (x,0), (y,0)), (q_1, (x,1), (y,0)), (q_2, (x,2), (y,1)), Act(x) = [0,2,1], Act(y) = [0,1])
b (x,2) ≤ 3,(y,1) ≤ 2

Figure 3 Steps in the construction of $\mathbb{A}$ corresponding to our running example. With the color coding in q'_1, q'_2, it is easy to see that q'_2 is same as q'_1 on removing the circled entities in q'_2. Same with q'_4 and q'_6.

all the functions of the form $v : X \mapsto \{0, \ldots, |Q| - 1\}$. Let $\mathcal{L}$ be a set of all the functions from Q to $V \cup \{0\}$. Let Active be a set of all the functions from X to a sequence (without duplicate) over $\{0, 1 \ldots |Q - 1|\}$. Then $\mathcal{Q} = \mathcal{L} \times \mathsf{Active}$. Intuitively, we replace the bunch of conjunctive transitions C into a single transition, similar to the subset construction for converting Alternating Finite Automata (AFA) to Non-Deterministic Finite Automata (NFA). But notice that we can have clauses (or conjunctions) of the form $q \wedge X'.q'$. Hence, simple subset construction won't work as we need to spawn multiple copies of clocks in X', wherein one of the elements of the new location $\{q, q'\}$ they are reset while in another they are not. In general, there could be an unbounded number of such clock copies required for a single clock, $x \in X$. But due to Lemma 14, if we make sure to compress the states (and hence remove redundant clocks), we need to keep at most $|Q|$ copies for each clock in X. In principle, we are constructing an NTA $\mathbb{A}$ whose transition system $TS(\mathbb{A})$ is simulation equivalent to the LTS $\mathsf{T}_{red}(\mathcal{A})$ (see the full version Proposition 19) and hence to input VWATA$^{0,\infty}$ $TS(\mathcal{A})$. Thus, by Proposition 6, $L(\mathcal{A}) = L(\mathbb{A})$. We present the idea via our running example.

3.2.4 Construction on Running Example

Please refer to the VWATA$^{0,\infty}$ of our running example Figure 1. We now illustrate the construction on our running example. We start with location q_0, with the 0^{th} copies of clock x and y. Hence

$$q'_0 = \{(q_0, (x, 0)(y, 0)), \mathsf{Active}(x) = [0], \mathsf{Active}(y) = [0]\}.$$

This corresponds to the configuration $C_0 = \{(q_0, \mathbf{0}_X)\}$ of $\mathcal{A}$. In the input automaton, the transitions from q_0 on a is defined by $\delta(q_0, a) = q_0 \wedge x.q_1$. Hence, we need to spawn a new copy of clock x as it is reset in one transition and not in another. We associate this new copy of clock x with the branch that resets x, i.e., this new clock x is associated with location q_1. Hence, we have $\Delta(q'_0, a) = (x, 1).q'_1$ where $q'_1 = \{(q_0, (x, 0), (y, 0)), (q_1, (x, 1)(y, 0)), \mathsf{Active}(x) = 0 \geq 1, \mathsf{Active}(y) = 0\}$. Intuitively, q'_1 corresponds to the configurations of the form $C_1 = \{(q_0, \nu^1_0), (q_1, \nu^1_1)\}$ of $\mathcal{A}$, where $\nu^1_0(x) =$ value of $(x, 0)$, $\nu^1_1(x) =$ value of $(x, 1)$, and

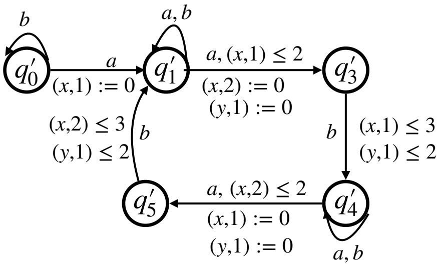


Figure 4 Final Automata after applying the reductions.

$\nu_0^1(y) = \nu_1^1(y) =$ value of $(y,0)$. We continue with this new location. Hence, we will consider the transitions from both q_0 and q_1 on a. The component $((q_0,(x,0),(y,0)))$ on a again spawns a new copy of clock x as it resets the clock in one while not resetting on self-loop, hence, getting $\{(q_0,(x,0),(y,0)),(q_1,(x,2)(y,0))\}$ (possibility 1 from q_0, the only possibility). Notice that we spawned $(x,2)$ as $(x,1)$ is in use by q_1 already. The component $(q_1,(x,1)(y,0))$ will be computed using the transition function of input automaton, i.e. $\delta(q_1,a) = (y.q_2 \wedge x \leq 2) \vee q_1$. Here, we either stay at q_1 with the same set of clock copies as before (possibility 1 from q_1), or we need a new copy of y while simultaneously checking for the clock copy of x corresponding to location q_1 (i.e. $(x,1)$) is ≤ 2 (possibility 2 from q_1). Combining the possibilities 1 from q_0 and q_1 we get, $\{(q_0,(x,0),(y,0),(q_1,(x,2),(y,0))(q_1,(x,1),y,0),\mathsf{Active}(x)=[0\geq 1\geq 2],\mathsf{Active}(y)=[0]\}$. But $q \in \mathsf{Q}^{\leq}$. Hence, if we can reach the accepting state from $(q_1,(x,1),(y,0))$ then we can reach the accepting state from $(q_1,(x,2),(y,0)$ too, as value of $(x,1) \geq$ value of $(x,2)$ (this fact is also encoded in the $\mathsf{Active}(x)$ sequence). Thus, $(q_1,(x,2),(y,0))$ can be removed from the new location without affecting simulation equivalence (and hence language equivalence). This corresponds to the removal of redundant states in the construction of the runs of $\mathsf{T}_{red}(\mathcal{A})$ from $T(\mathcal{A})$. Hence, after deletion we get $q'_2=\{(q_0,(x,0),(y,0),\cancel{(q_1,(x,2),(y,0))},(q_1,(x,1),y,0),\mathsf{Active}(x)=[0\geq 1\cancel{\geq 2}],\mathsf{Active}(y)=[0]\}=q'_1$.

Thus, combining result of the transition of q_0 on a and possibility 1 from q_1 we get $q'_1=\{q_0,(x,0),(y,0),(q_1,(x,1),y,0),\mathsf{Active}(x)=0\geq 1,\mathsf{Active}(y)=0\}$.
Combining results possibility 1 from q_0 and possibility 2 from q_1, we get $\{q_0,(x,0),(y,0),(q_1,(x,2),(y,0)),\ (q_2,(x,1),(y,1))\mathsf{Active}(x)=0\geq 2\geq 1,\mathsf{Active}(y)=0\geq 1.\}=q'_3$ if $(x,1) \leq 2$. Note that each location from Q appears at most once in q'_3. Hence, there is no scope of reduction. Combining the above two combination of possibilities, $\Delta(q'_1,a) = q'_1 \vee (y,1).(q'_3 \wedge x \leq 2)$. Continuing this we get the resulting NTA $\mathbb{A}$ equivalent to the input formula ϕ. Notice that we are eliminating the conjunctive transitions using subset like construction and keeping the disjunctions as it is. Hence, after eliminating all the conjunctive transitions the reduced automata contains only disjunctions amongst different locations in the output formulae of the transitions giving an NTA. Refer to Figures 3, 4.

3.2.5 Worst Case Complexity

By construction in [2], the number of states in the region automata of $\mathbb{A} = W \leq |\mathcal{Q}| \times (|X| \times |Q|)! \times 2 \times (c_{max}+1)$ where c_{max} is the max constant used in the guards in $\mathcal{G}$ and $|\mathcal{Q}| = |Q| \times (|X|^{|Q|}+1) \times (|Q|!)^{|X|}$. Hence, $W = O(2^{Poly(|A|)})$ implying that the emptiness could be checked in NPSPACE = PSPACE . Notice that the state containing the location (L,Act) will only have to store the region information of active clocks, which, in practice, could be much less than the worst case. Hence, lazily spawning clock copies may result in NTA with much less number of clocks than the worst case (i.e. $|X| \times |Q|$).

4 Expressiveness of TPTL$^{0,\infty}$

We now compare the expressive power of 1-TPTL$^{0,\infty}$ with respect to that of MITL.

4.1 Metric Temporal Logic(MTL)

MTL is a real-time extension of LTL where the U modality is guarded with an interval. Syntax of MTL is defined as follows. $\varphi ::= a \,|\top\, |\varphi \wedge \varphi \mid \neg\varphi \mid \varphi \mathsf{U}_I \varphi$,
where $a \in \Sigma$ and $I \in \mathcal{I}_{\mathsf{int}}$. For a timed word $\rho = (\sigma_1, \tau_1)(\sigma_2, \tau_2)\dots(\sigma_n, \tau_n) \in T\Sigma^*$, a position $i \in dom(\rho)$, an MTL formula φ, the satisfaction of φ at a position i of ρ, denoted $\rho, i \models \varphi$, is defined as follows. We discuss only the semantics of temporal modalities. Boolean operators mean as usual. $\rho, i \models \varphi_1 \mathsf{U}_I \varphi_2$ iff $\exists j > i. \rho, j \models \varphi_2, \tau_j - \tau_i \in I$, and $\forall i < k < j. \rho, k \models \varphi_1$. As usual, $\mathsf{F}_I(\phi) = \top \mathsf{U}_I \phi$, $\mathsf{G}(\phi) = \neg \mathsf{F}_I \neg\phi$, $\mathsf{Next}_I \phi = \bot \mathsf{U}_I \phi$. The language of an MTL formula φ is defined as $L(\varphi) = \{\rho | \rho, 1 \models \varphi\}$. The subclass of MTL where the intervals I in the "until" modalities are restricted to be **non-punctual** is known as Metric Interval Temporal Logic (MITL) . MITL$^{0,\infty}$ [1, 3, 13] is the subclass of MTL where intervals are restricted in $\mathcal{I}^{0,\infty}_{\mathsf{int}}$. Satisfiability Checking for MITL (MITL$^{0,\infty}$) is ExpSpace -complete (PSpace -complete) [4, 1, 3]. MITL is strictly more expressive than MITL$^{0,\infty}$ in pointwise semantics [12].

▶ Remark 15. Any MTL formula can be translated to an equivalent 1-TPTL (closed) formula using the following equivalence recursively. $\varphi_1 \mathsf{U}_I \varphi_2 \equiv x.(\varphi_1 \mathsf{U} \varphi_2 \wedge x \in I)$.

4.2 Expressiveness of TPTL$^{0,\infty}$

▶ **Theorem 16.** *1-TPTL$^{0,\infty}$ is strictly more expressive than MITL.*

Proof. Both MITL and 1-TPTL$^{0,\infty}$ are closed under all boolean operations. Hence, we just need to show that any formula of the form $\varphi' \mathsf{U}_I \varphi$ is expressible in 1-TPTL$^{0,\infty}$. Notice that, any MITL formula $\varphi' \mathsf{U}_{[l,u)} \varphi \equiv [\mathsf{G}_{[0,l)}\{\varphi' \wedge (\varphi' \mathsf{U} \varphi)\}] \wedge [\mathsf{F}_{[l,l+1)}\varphi \vee \mathsf{F}_{[l+1,l+2)}\varphi \dots \mathsf{F}_{[u-1,u)}\varphi]$. (similar reduction applies for other kinds of intervals). $\mathsf{G}_{[0,l)}(\varphi' \wedge (\varphi' \mathsf{U} \varphi))$ is already in MITL$^{0,\infty}$ (and hence in 1-TPTL$^{0,\infty}$ by remark 15). Hence, it suffices to encode modalities of the form $\mathsf{F}_{[l,l+1)}$ using 1-TPTL$^{0,\infty}$ formula. Let $\rho = (a_1, \tau_1), (a_2, \tau_2) \dots$ be any timed word. Let $i \in dom(\rho)$ be any point. $\rho, i \models \mathsf{F}_{[l,l+1)}(\varphi)$ iff there exists a point $i' > i$ such that $\tau_{i'} - \tau_i \in [l, l+1)$ and $\rho, i' \models \varphi$. ρ has a point i' within $[l, l+1)$ interval from i where φ holds iff there exist earliest such point j ($j \le i'$) within $[l, l+1)$ from i where φ holds iff there is a point $j' > i$ such that $\tau_{j'} - \tau_i \ge l$ (i.e. $\rho, i \models \phi_0 = \mathsf{F}_{[l,\infty)}\varphi$), and let j be the first point such that $\tau_j - \tau_i \ge l$, and $\rho, j \models \varphi$. Such a point exists due to ϕ_0. Then:

- Case 1: Either there is no point strictly between i and j where φ holds. Then occurrence of j within $l + 1$ can be expressed using formula, $\phi_1 = \neg \mathsf{F}_{[0,l)}\varphi \wedge \mathsf{F}_{[0,l+1)}\varphi$.
- Case 2: Or there exists a point k such that $\tau_j - \tau_k < 1$, $\tau_k - \tau_i \in [l-1, l)$, and $\rho, k \models \varphi$. Equivalently, i satisfies $\phi_2 = \mathsf{G}_{[l-1,l)}(\mathsf{F}_{[0,1]}(\varphi))$,
- Case 3: Or there exists a point k with $i < k < j$ such that $\tau_j - \tau_k \ge 1$, $\rho, k \models \varphi$, and $\forall k < k' < j. \rho, k' \not\models \varphi$.

  (1) Such a point k satisfies $\phi_{\mathsf{approach}} = \varphi \wedge \mathsf{G}_{[0,1)}(\neg\varphi)$. Indeed a key property is that k, the last point in $[0, l)$ satisfying ϕ, satisfies ϕ_{approach}. By the definition of k, i.e., there are no occurrences of φ after k in $[0, l)$.

  (2) Notice that any two point k_1 and k_2 satisfying ϕ_{approach} are at least a unit time apart. Hence, there could be at most l points satisfying ϕ_{approach} within $[0, l)$. Then, the following 1-TPTL$^{0,\infty}$ formula $\mathsf{Count}\phi_{\mathsf{approach}}(n)$ with parameter n states that there are exactly n points, $1 \le n \le l$ within $[0, l)$ of point i where ϕ_{approach} holds. Here, $\mathsf{Count}\phi_{\mathsf{approach}}(n) = \phi_{\ge n} \wedge \neg\phi_{\ge n+1}$, where $\phi_{\ge n} = x.((\neg\phi_{\mathsf{approach}})\mathsf{U}(\phi_{\mathsf{approach}} \wedge ((\neg\phi_{\mathsf{approach}})\mathsf{U}(\phi_{\mathsf{approach}} \wedge \underbrace{\dots}_{n-3} \wedge((\neg\phi_{\mathsf{approach}})\mathsf{U}(\phi_{\mathsf{approach}} \wedge x < l)\dots)))$.

Observe that for a given timed word and interval $[0, l]$ from i, there is a unique n satisfying this formula $\mathsf{Count}\phi_{\mathsf{approach}}(n)$.

(3) Using this n, the formula $\gamma(n, \varphi) = x.(\neg\phi_{\mathsf{approach}}\mathsf{U}(\phi_{\mathsf{approach}} \wedge \neg\phi_{\mathsf{approach}}\mathsf{U}(\phi_{\mathsf{approach}} \wedge \underbrace{\ldots}_{n-3} \wedge((\neg\phi_{\mathsf{approach}})\mathsf{U}(\phi_{\mathsf{approach}} \wedge \mathsf{G}(x \leq l \vee \neg\varphi) \wedge \mathsf{F}(\varphi \wedge x < l + 1))\ldots))$ holds if after n occurrences of ϕ_{approach} (which gives point k), the next occurrence of φ occurs before time $l + 1$.

Hence, case 3 is characterized by the formula $\phi_3 = \bigvee_{n=1}^{l} \mathsf{Count}\phi_{\mathsf{approach}}(n) \wedge \gamma(n, \varphi)$.

Hence, the required formula $\psi = \phi_0 \wedge (\phi_1 \vee \phi_2 \vee \phi_3)$.

For strict containment of MITL, consider the formula $\beta = x.\mathsf{F}(b \wedge \mathsf{F}(b \wedge x \leq 1))$. This specifies, there exist at least two points within the next unit interval where b holds. [15, 29, 23] show that this formula is not expressible even in MTL. ◀

5 Discussion and Conclusion

Ferrère [8] proposed an extension of LTL with Metric Interval Regular Expressions called Metric Interval Dynamic Logic (MIDL) and showed it to be more expressive than EMITL of [35]. We claim that our proof of PSpace completeness for 1-ATA$^{0,\infty}$ emptiness implies the same for MIDL$_{0,\infty}$ satisfiability strictly generalizing the results and techniques of [16] which proved the same for EMITL$_{0,\infty}$. This resolves one of the "future directions" of [16].

Authors in [19] generalized the notion of non-punctuality to non-adjacency for 1-TPTL. We remark that unfortunately, this notion doesn't help in making 2-TPTL decidable. Notice that $\varphi = \mathsf{G}x.\{\neg\phi \vee \mathsf{F}y.(\top \wedge x \in [1, 2] \wedge \mathsf{F}(\phi_1 \wedge x \in [1, 2] \wedge y \in [1, 2]))\} \equiv \mathsf{G}[\phi \to \mathsf{F}_{[1,1]}(\mathsf{F}_{[1,1]}\phi_1)]$. Because, for any point i where φ holds there is a point j in the future such that $\tau_j - \tau_i \in [1, 2]$, and from that point j there is a point k in the future where ϕ_2 holds such that $\tau_k - \tau_j \in [1, 2]$ and $\tau_k - \tau_i \in [1, 2]$. Solving the inequalities we get, $\tau_j - \tau_i = 1$ and $\tau_k - \tau_i = 2$. Hence, φ can express some restricted form of punctual timing properties which leads to the undecidability of satisfiability using encoding similar to [25].

MITL$^{0,\infty}$ was extended with Counting (TLC) and Pnueli (TLP) modalities by [15] to increase the expressiveness, meanwhile maintaining the decidability in EXPSPACE and PSPACE, respectively. These logics TLP and TLC have the same expressive power in continuous semantics. While these logics were strictly more expressive than MITL in continuous semantics, in pointwise semantics they are incomparable. This is due to inexpressivity of arbitrary non-punctual metric constraints using unilateral metric interval constraints in pointwise semantics (see [16]). However, TLP and TLC properties are trivially expressible in TPTL$^{0,\infty}$ (one clock and nested until), making our logic strictly more expressive than these. As one of our future works, we would like to show that TLCI and TLPI (extensions of TLP and TLC using arbitrary non-punctual intervals) which are decidable in EXPSACE are expressible in TPTL$^{0,\infty}$.

Finally, we leave open (i) the extension of this work with Past modalities, (ii) FOL-like characterizations of TPTL$^{0,\infty}$, and (iii) whether adding multiple clocks in TPTL$^{0,\infty}$ improves expressiveness.

References

1 R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *J.ACM*, 43(1):116–146, 1996.

2 Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

3 Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In Luigi Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 139–152. ACM, 1991. doi:10.1145/112600.112613.

4 Rajeev Alur and Thomas A. Henzinger. Back to the future: Towards a theory of timed regular languages. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 177–186. IEEE Computer Society, 1992. doi:10.1109/SFCS.1992.267774.

5 Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. *Inf. Comput.*, 104(1):35–77, 1993. doi:10.1006/inco.1993.1025.

6 Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, January 1994. doi:10.1145/174644.174651.

7 Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of tptl and mtl. In Sundar Sarukkai and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, pages 432–443, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

8 Thomas Ferrère. The compound interest in relaxing punctuality. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink, editors, *Formal Methods – 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2018. doi:10.1007/978-3-319-95582-7_9.

9 Paul Gastin and Denis Oddoux. LTL with past and two-way very-weak alternating automata. In Branislav Rovan and Peter Vojtás, editors, *Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003, Proceedings*, volume 2747 of *Lecture Notes in Computer Science*, pages 439–448. Springer, 2003. doi:10.1007/978-3-540-45138-9_38.

10 Christoph Haase, Joël Ouaknine, and James Worrell. On process-algebraic extensions of metric temporal logic. In A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 283–300. Springer, 2010. doi:10.1007/978-1-84882-912-1_13.

11 Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly ltl model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 191–205, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

12 Thomas A. Henzinger. It's about time: Real-time logics reviewed. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, pages 439–454, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

13 Thomas A. Henzinger, Jean-François Raskin, and Pierre-Yves Schobbens. The regular real-time languages. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 580–591. Springer, 1998. doi:10.1007/BFb0055086.

14 Y. Hirshfeld and A. Rabinovich. An expressive temporal logic for real time. In *MFCS*, pages 492–504, 2006.

15 Yoram Hirshfeld and Alexander Rabinovich. Expressiveness of metric modalities for continuous time. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science – Theory and Applications*, pages 211–220, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

16 Hsi-Ming Ho. Revisiting timed logics with automata modalities. In Necmiye Ozay and Pavithra Prabhakar, editors, *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 67–76. ACM, 2019. doi:10.1145/3302504.3311818.

17 James Jerson Ortiz, Axel Legay, and Pierre-Yves Schobbens. Memory event clocks. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, pages 198–212, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

18 S. N. Krishna K. Madnani and P. K. Pandya. On unary fragments of mtl over timed words. In *ICTAC*, pages 333–350, 2014.

19 Shankara Narayanan Krishna, Khushraj Madnani, Manuel Mazo Jr., and Paritosh K. Pandya. Generalizing non-punctuality for timed temporal logic with freeze quantifiers. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods – 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2021. `doi:10.1007/978-3-030-90870-6_10`.

20 Shankara Narayanan Krishna, Khushraj Madnani, and Paritosh K. Pandya. Logics meet 1-clock alternating timed automata. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 39:1–39:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.CONCUR.2018.39`.

21 Slawomir Lasota and Igor Walukiewicz. Alternating timed automata. *ACM Trans. Comput. Log.*, 9(2):10:1–10:27, 2008. `doi:10.1145/1342991.1342994`.

22 Christof Loding and Wolfgang Thomas. Alternating automata and logics over infinite words. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, pages 521–535, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

23 Khushraj Nanik Madnani. *On Decidable Extensions of Metric Temporal Logic*. PhD thesis, Indian Institute of Technology Bombay, Mumbai, India, 2019.

24 J. Ouaknine and J. Worrell. On the decidability of metric temporal logic. In *LICS*, pages 188–197, 2005.

25 J. Ouaknine and J. Worrell. Safety metric temporal logic is fully decidable. In *TACAS*, pages 411–425, 2006.

26 Paritosh K. Pandya and Simoni S. Shah. On expressive powers of timed logics: Comparing boundedness, non-punctuality, and deterministic freezing. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory – 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2011. `doi:10.1007/978-3-642-23217-6_5`.

27 Paritosh K. Pandya and Simoni S. Shah. The unary fragments of metric interval temporal logic: Bounded versus lower bound constraints. In *Automated Technology for Verification and Analysis – 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, pages 77–91, 2012.

28 Radek Pelánek and Jan Strejček. Deeper connections between ltl and alternating automata. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *Implementation and Application of Automata*, pages 238–249, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

29 A. Rabinovich. Complexity of metric temporal logic with counting and pnueli modalities. In *FORMATS*, pages 93–108, 2008.

30 Alexander Rabinovich. Complexity of metric temporal logics with counting and the pnueli modalities. *Theor. Comput. Sci.*, 411(22-24):2331–2342, 2010. `doi:10.1016/j.tcs.2010.03.017`.

31 Jean Francois Raskin. *Logics, Automata and Classical Theories for Deciding Real Time*. PhD thesis, Universite de Namur, 1999.

32 Gareth Scott Rohde. *Alternating Automata and the Temporal Logic of Ordinals*. PhD thesis, University of Illinois at Urbana-Champaign, USA, 1997. AAI9812757.

33 Heikki Tauriainen. *Automata and linear temporal logic: Translations with transition-based acceptance*. Doctoral thesis, Helsinki University of Technology, 2006.

34 Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, pages 238–266, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. doi:10.1007/3-540-60915-6_6.

35 Thomas Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS, Lübeck, Germany, September 19-23, Proceedings*, pages 694–715, 1994. doi:10.1007/3-540-58468-4_191.

Model-Checking Parametric Lock-Sharing Systems Against Regular Constraints

Corto Mascle
Université de Bordeaux, France

Anca Muscholl
Université de Bordeaux, France

Igor Walukiewicz
Université de Bordeaux, CNRS, France

Abstract

In parametric lock-sharing systems processes can spawn new processes to run in parallel, and can create new locks. The behavior of every process is given by a pushdown automaton. We consider infinite behaviors of such systems under strong process fairness condition. A result of a potentially infinite execution of a system is a limit configuration, that is a potentially infinite tree. The verification problem is to determine if a given system has a limit configuration satisfying a given regular property. This formulation of the problem encompasses verification of reachability as well as of many liveness properties. We show that this verification problem, while undecidable in general, is decidable for nested lock usage.

We show EXPTIME-completeness of the verification problem. The main source of complexity is the number of parameters in the spawn operation. If the number of parameters is bounded, our algorithm works in PTIME for properties expressed by parity automata with a fixed number of ranks.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Parametric systems, Locks, Model-checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.24

Related Version *Full Version*: https://arxiv.org/abs/2307.04925

Funding Work supported by the ANR project PaVeDys.

1 Introduction

Locks are a widely used concurrency primitive. They appear in classical programming languages such as Java, as well as in recent ones such as Rust. The principle of creating shared objects and protecting them by mutexes (like the "synchronized" paradigm in Java) requires *dynamic lock creation.* The challenge is to be able to analyze programs with dynamic creation of threads *and* locks.

Our system model is based on Dynamic Pushdown Networks (DPNs) as introduced in [7], where processes are pushdown systems that can spawn new processes. The DPN model was extended in [20] by adding synchronization through a fixed number of locks. Here we take a step further and allow dynamic lock creation: when spawning a new process, the parent process can pass some of its locks, and new locks can be created for the new thread. This way we model recursive programs with creation of threads and locks. We call such systems *dynamic lock-sharing systems* (DLSS).

The focus in both [7] and [20] is computing the *Pre** of a regular set of configurations, and they achieve this by extending suitably the saturation technique from [6]. Here we consider not only reachability but also infinite behaviors of DLSS under fairness conditions. For this we propose a different approach than saturation from [7, 20] as saturation is not suited to cope with liveness properties.

We show that verifying regular properties of DLSS is decidable if every process follows *nested lock usage*. This means that locally every process acquires and releases the locks according to the stack discipline. Nested locking is assumed in most papers on parametric verification of systems with synchronization over locks. It is also considered as good programming practice, sometimes even enforced syntactically, as in Java through synchronized blocks.

Without any restriction on lock usage we show that our problem is undecidable, even for finite state processes and reachability properties that refer to a single process. Note that our model does not have global variables. It is well-known that reachability is undecidable already for two pushdown processes with one lock and one global variable.

Outline of the paper. Our starting point is to use trees to represent configurations of DLSS. This representation was introduced in [20]. The advantage is that it does not require to talk about interleavings of local runs of processes. Instead it represents every local run as a left branch in a tree and the spawn operations as branching to the right. At each computation step one or two nodes are added below a leaf of the current configuration. Thus, the result of a run of DLSS is an infinite tree that we call a *limit configuration*. Our first observation is that it is easy to read out from a limit configuration of a run if the run is strongly process-fair (Proposition 3).

An important step is to characterize those trees that are limit configurations of runs of a given *finite state* DLSS, namely where every process is a finite state system. This is done in Lemma 11. To deal with lock creation this lemma refers to the existence of some global acyclic relation on locks. We show that this global relation can be recovered from local orderings in every node of the configuration tree (Lemma 12). Finally, we show that there is a nondeterministic Büchi tree automaton verifying all the conditions of Lemmas 11 and 12. This is the desired tree automaton recognizing limit configurations of process-fair runs. Our verification problem is solved by checking if there is a tree satisfying the specification and accepted by this automaton. This way we obtain the upper bound from Theorem 7. Surprisingly the size of the Büchi automaton is linear in the size of DLSS, and exponential only in the *arity* of the DLSS, which is the maximal number of locks a process can access. For example, in the dining philosophers setting (cf. Figure 1) the arity is 3, as every philosopher has access only to its left and right forks, implemented as locks; and there is one more fork to close the cycle.

The extension of our construction to pushdown processes requires one more idea to get an optimal complexity. In this case, ensuring that the limit tree represents a computation requires using pushdown automata. Hence, the Büchi tree automaton as described in the previous paragraph becomes a pushdown Büchi automaton on trees. The emptiness of pushdown Büchi tree automata is EXPTIME-complete, which is an issue as the automaton constructed is already exponential in the size of the input. However, we observe that the automata we obtain are right-resetting, since new threads are spawned with empty pushdown. Intuitively, the pushdown is needed only on left paths of the configuration tree to check correctness of local runs. A right-resetting automaton resets its stack each time it goes to the right child. We show that the emptiness of right-resetting parity pushdown tree automata can be checked in PTIME if the biggest rank in the parity condition is fixed (if it is not fixed then the problem is at least as complex as solving parity games). This gives the upper bound from Theorem 8.

Finally, we obtain the matching lower bound by proving EXPTIME-hardness of checking if a process of the DLSS has an infinite run (Proposition 22). This holds even for finite state processes. We also show that even for finite state processes the DLSS verification problem is undecidable if we allow arbitrary usage of locks (Theorem 5).

Related work. Parametrized verification has remained an active research area for almost three decades [1, 5, 13]. It has brought a steady stream of works on parametric systems with locks. As already mentioned, the first directly relevant paper is [7] introducing Dynamic Pushdown Networks (DPNs). These consist of pushdown processes with spawn but no locks. The main idea is to represent a configuration as a sequence of process identifiers, each identifier followed by a stack content. Computing Pre^* of a regular set of configurations is decidable by extending the saturation technique from [6].

An important step is made in [20] where the authors introduce a tree representation of configurations. This is essentially the same representation as we use here. They extend DPNs by a fixed set of locks, and show how to adapt the saturation technique to compute Pre^* in this case. Their result is an EXPTIME decision procedure for verifying reachability of a regular set of configurations. This work has been extended to incorporate join operations [12], or priorities on processes [9]. Our work extends [20] in two directions: it adds lock creation, and considers liveness properties. It is not clear how one could extend saturation methods to deal with liveness properties.

The saturation method has been adapted to DPNs with lock creation in the recent thesis [17]. The approach relies on hyperedge replacement grammars, and gives decidability without complexity bounds. Our liveness conditions can express this kind of reachability conditions.

Actually, the first related paper to deal with lock creation is probably [25]. The authors consider a model of higher-order programs with spawn, joins, and lock creation. Apart from nested locking, a new restriction of scope safety is imposed. Under these conditions, reachability of pairs of states is shown to be decidable.

The works above have been followed by implementations [9, 18, 25]. In particular [9] reports on verification of several substantial size programs and detecting an error in xvisor [8].

In all the works above nested locking is assumed. In [16] the interest of nested locking is underlined by showing that reachability with two pushdown processes using locks is undecidable in general, but it is decidable for nested locking. There are only few related works without this assumption. The work [15] generalizes nested locking to bounded lock-chain condition, and shows decidability of reachability for two pushdown processes. In [19] the authors consider contextual locking where arbitrary locking may occur as long as it does not cross procedure boundaries. This condition is incomparable with nested locking.

Finally, we comment on shared state and global variables. These are not present in the above models because reachability for two pushdown processes with one lock and one global variable is already undecidable. There is an active line of study of multi-pushdown systems where shared state is modeled as global control. In this model decidability is recovered by imposing restrictions on stack usage such as bounded context switching and variations thereof [2, 22–24]. Observe that these are restrictions on global runs, and not on local runs of processes, as we consider here. Another approach to recover decidability is to have shared state but no locks [10, 11, 14, 21]. Finally, there is a very interesting model of threaded pools [3, 4], without locks, where verification is decidable once again assuming bounded context switching. But the complexity of this model is as high as Petri net coverability [4].

Structure of the paper. The next section presents the main definitions and results. The main proof for finite state processes is outlined is Sections 3 and 4. Section 5 describes the extension to pushdown processes. Missing proofs can be found in the appendix of the full version on arXiv.

2 Definitions and results

A dynamic lock-sharing system is a set of processes, each process has access to a set of locks and can spawn other processes. A spawned process can inherit some locks of the spawning process and can also create new locks. All processes run in parallel. A run of the system must be fair, meaning that if a process can move infinitely many times then it eventually does.

More formally, we start with a finite set of process identifiers Proc. Each process identifier $p \in \mathit{Proc}$ has an arity $ar(p) \in \mathbb{N}$ telling how many locks the process uses. The process can refer to these locks through the variables $\mathit{Var}(p) = \{x_1^p, \dots, x_{ar(p)}^p\}$. At each step a process can do one of the following operations:

$$\begin{aligned} \mathit{Op}(p) = &\{\mathtt{nop}\} \cup \{\mathtt{get}_x, \mathtt{rel}_x \mid x \in \mathit{Var}(p)\} \\ &\cup \{\mathtt{spawn}(q,\sigma) \mid q \in \mathit{Proc}, \sigma : \mathit{Var}(q) \to (\mathit{Var}(p) \cup \{\mathtt{new}\})\} \end{aligned}$$

Operation $\mathtt{nop}$ does nothing. Operation $\mathtt{get}_x$ acquires the lock designated by x, while $\mathtt{rel}_x$ releases it. Operation $\mathtt{spawn}(q,\sigma)$ spawns an instance of process q where every variable of q designates a lock determined by the substitution σ; this can be a lock of the spawning process or a new lock, if $\sigma(x^q) = \mathtt{new}$. We require that the mapping σ is *injective* on $\mathit{Var}(p)$. This is important for the definition of nested stack usage.

A *dynamic lock-sharing system* (DLSS for short) is a tuple

$$\mathcal{S} = (\mathit{Proc}, ar, (\mathcal{A}_p)_{p \in \mathit{Proc}}, p_{init}, \mathcal{L}ocks)$$

where Proc, and ar are as described above. For every process p, $\mathcal{A}_p$ is a transition system describing the behavior of p. Process $p_{init} \in \mathit{Proc}$ is the initial process. Finally, $\mathcal{L}ocks$ is an infinite pool of locks.

Each transition system $\mathcal{A}_p$ is a tuple $(S_p, \Sigma_p, \delta_p, op_p, init_p)$ with S_p a finite set of states, $init_p$ the initial state, Σ_p a finite alphabet, $\delta_p : S_p \times \Sigma_p \to S_p$ a *partial* transition function, and $op_p : \Sigma_p \to \mathit{Op}(p)$ an assignment of an operation to each action. We require that the Σ_p are pairwise disjoint, and define $\Sigma = \bigcup_{p \in \mathit{Proc}} \Sigma_p$. We write $op(b)$ instead of $op_p(b)$ for $b \in \Sigma_p$, as b determines the process p.

For simplicity, we require that p_{init} is of arity 0. In particular, process p_{init} has no $\mathtt{get}$ or $\mathtt{rel}$ operations.

An example in Figure 1 presents a DLSS modeling an arbitrary number of dining philosophers. The system can generate a ring of arbitrarily many philosophers, but can also generate infinitely many philosophers without ever closing the ring.

A configuration of $\mathcal{S}$ is a tree representing the runs of all active processes. The leftmost branch represents the run of the initial process p_{init}, the left branches of nodes to the right of the leftmost branch represent runs of processes spawned by p_{init} etc. So a leaf of a configuration represents the current situation of a process that is started at the first ancestor above the leaf that is a right child. A node of a configuration is associated with a process, and tells in what state the process is, which locks are available to it, and which of them it holds.

More formally, a *configuration* is a, possibly infinite, tree $\tau \subseteq \{0,1\}^*$, with each node ν labeled by a tuple (p, s, a, L, H) where $p \in \mathit{Proc}$ is the process executing in ν, $s \in \Sigma_p$ the state of p, $a \in \Sigma_p$ the action p executed at ν, or $\perp \notin \Sigma$ if ν is a root, $L : \mathit{Var}(p) \to \mathcal{L}ocks$ an assignment of locks to variables of p, and $H \subseteq L(\mathit{Var}(p))$ the set of locks p holds before executing a. We use $p(\nu)$, $s(\nu)$, $a(\nu)$, $L(\nu)$ and $H(\nu)$ to address the components of the label of ν. For ease of notation we will write $\mathit{Var}(\nu)$ instead of $\mathit{Var}(p(\nu))$.

$$\begin{aligned}
p_{init}: &\quad \mathtt{spawn}(\mathit{first}, \mathtt{new}, \mathtt{new}) \\
\mathit{first}(x_l, x_r): &\quad \mathtt{spawn}(\mathit{phil}, x_l, x_r); \mathtt{spawn}(\mathit{next}, x_r, \mathtt{new}, x_l) \\
\mathit{next}(x_l, x_r, x_{\mathrm{lfirst}}): &\quad \text{or} \begin{cases} \mathtt{spawn}(\mathit{phil}, x_l, x_{\mathrm{lfirst}}) \\ \mathtt{spawn}(\mathit{phil}, x_l, x_r); \mathtt{spawn}(\mathit{next}, x_r, \mathtt{new}, x_{\mathrm{lfirst}}) \end{cases} \\
\mathit{phil}(x_l, x_r): &\quad \text{repeat-forever or} \begin{cases} \mathtt{get}_{x_l}; \mathtt{get}_{x_r}; \mathrm{eat}; \mathtt{rel}_{x_r}; \mathtt{rel}_{x_l} \\ \mathtt{get}_{x_r}; \mathtt{get}_{x_l}; \mathrm{eat}; \mathtt{rel}_{x_l}; \mathtt{rel}_{x_r} \end{cases}
\end{aligned}$$

Figure 1 Dining philosophers: process *first* starts the first philosopher and an iterator process *next* starts successive philosophers. The forks, modeled as locks, are passed via variables x_l and x_r. The third variable x_{lfirst} of *next* is the left fork of the first philosopher used also by the last philosopher. The system is nested as *phil* takes and releases forks in the stack order. The arity of the system is 3.

We write $H(\tau)$ for the set of locks *ultimately held* by some process in τ, that is, $H(\tau) = \{\ell : \text{for some } \nu,\ \ell \in H(\nu') \text{ for all } \nu' \text{ on the leftmost path from } \nu\}$. If τ is finite this is the same as to say that $H(\tau)$ is the union of $H(\nu)$ over all leaves ν of τ.

The initial configuration is the tree τ_{init} consisting only of the root ε labeled by $(p_{init}, init_p, \perp, \emptyset, \emptyset)$. Suppose that ν is a leaf of τ labeled by (p, s, b, L, H), and there is a transition $s \xrightarrow{a} s'$ for some s' in $\mathcal{A}_p$. A transition between two configurations $\tau \xrightarrow{\nu,a} \tau'$ is defined by the following rules.

- If $op(a) = \mathtt{spawn}(q, \sigma)$ then τ' is obtained from τ by adding two children $\nu 0, \nu 1$ of ν. The label of the left child $\nu 0$ is (p, s', a, L, H). The label of the right child $\nu 1$ is $(q, init_q, \perp, L', \emptyset)$ where $L'(x^q) = L(\sigma(x^q))$ if $\sigma(x^q) \neq \mathtt{new}$ and $L'(x^q) = \ell_{\nu, x^q}$ is a fresh lock, otherwise.
- Otherwise, τ' is obtained from τ by adding a left child $\nu 0$ to ν. The label of $\nu 0$ must be of the form (p, s', a, L, H') subject to the following constraints:
  - If $op(a) = \mathtt{nop}$ then $H' = H$,
  - If $op(a) = \mathtt{get}_x$ and $L(x) \notin H(\tau)$ then $H' = H \cup \{L(x)\}$,
  - If $op(a) = \mathtt{rel}_x$ and $L(x) \in H$ then $H' = H \setminus \{L(x)\}$.

  Note that we do not allow a process to acquire a lock it already holds, or release a lock it does not have. We call this property *soundness*.

A *run* is a (finite or infinite) sequence of configurations $\tau_0 \xrightarrow{\nu_1, a_1} \tau_1 \xrightarrow{\nu_2, a_2} \cdots$. As the trees in a run are growing we can define the *limit configuration* of that run as its last configuration if it is finite, and as the limit of its configurations if it is infinite.

▶ Remark 1. Note that in a run, at every moment distinct variables of a process are associated with distinct locks: $L(\nu_i)(x) \neq L(\nu_i)(y)$ for all $x, y \in Var(\nu_i)$ with $x \neq y$.

▶ Remark 2. The labels L and H can be computed out of the other three labels in the tree just following the transition rules. We could have defined configurations as trees with only three labels (p, s, a), but we preferred to include L and H for readability. Yet, later we will work with tree automata recognizing configurations and there it will be important that the labels come from a finite set.

A configuration τ is *fair* if for no leaf ν there is a transition $\tau \xrightarrow{\nu,a} \tau'$ for some a and τ'. We show that this compact definition of fairness captures strong process fairness of runs. Recall that a run is *strongly process-fair* if whenever from some position in the run a process is enabled infinitely often then it moves after this position.

▶ **Proposition 3.** *Consider a run $\tau_0 \xrightarrow{\nu_1,a_1} \tau_1 \xrightarrow{\nu_2,a_2} \cdots$ and its limit configuration τ. The run is strongly process-fair if and only if τ is fair.*

Objectives. Instead of using some specific temporal logic we stick to a most general specification formalism and use regular tree properties for specifications. A *regular objective* is given by a nondeterministic tree automaton $\mathcal{B}$ over $\Sigma \cup \{\perp\}$, which defines a language of accepted limit configurations. The trees we work with can have nodes of rank 0, 1, or 2. So we suppose that the alphabet is partitioned into Σ_0, Σ_1 and Σ_2. The nondeterministic transition function reflects this with $\delta(q,a) \subseteq \{\top\}$ if $a \in \Sigma^0$, $\delta(q,a) \subseteq Q$ if $a \in \Sigma_1$, and $\delta(q,a) \subseteq Q \times Q$ if $a \in \Sigma_2$. A run of the automaton on a tree t is a labeling of t with states respecting δ. In particular if ν is a leaf of t then $\top \in \delta(q,a)$, where q is the state and a is the letter in ν. A run is accepting if for every infinite path the sequence of states on this path is in the accepting set of the automaton. We will work with accepting sets given by parity conditions. We say that a configuration τ satisfies $\mathcal{B}$ when $\mathcal{B}$ accepts the tree obtained from τ by restricting only to action labels.

Regular objectives can express many interesting properties. For example, "for every instance of process p its run is in a regular language $\mathcal{C}$". Or more complicated "there is an instance of p with a run in a regular language $\mathcal{C}_1$ and all the instances of p have runs in the language $\mathcal{C}_2$". Of course, it is also possible to talk about boolean combinations of such properties for different processes. Observe that the resulting automaton $\mathcal{B}$ for these kinds of properties can be a parity automaton with ranks $1,2,3$ (properties of sequences can be expressed by Büchi automata, and rank 3 is used to implement existential quantification on process instances).

Regular objectives can express deadlock properties. Since we only consider process-fair runs, a finite branch in a limit configuration indicates that a process is blocked forever after some point. Hence, we can express properties such as "there is an instance of p that is blocked forever after a finite run in a regular language $\mathcal{C}$". We can also express that all branches are finite, which is equivalent to a global deadlock since we are considering only process-fair runs.

Reachability properties are also expressible with regular objectives. We can check simultaneous reachability of several states in different branches, for instance "there is a reachable configuration in which some process p reaches s while some process p' reaches s'". There are ways to do it directly, but the shortest argument is through a small modification of the DLSS. We can simply add transitions to stop processes non-deterministically in desired states: adding new `nop` transitions from s and s' to new deadlock states. Using ideas from [19] we can also check reachability of a regular set of configurations.

Going back to our dining philosophers example from Figure 1, we can see also other types of properties we would like to express. For example, we would like to say that there are finitely many philosophers in the system. This can be done simply by saying that there are not infinitely many spawns in the limit configuration. (In this example it is equivalent to saying that there is no branch turning infinitely often to the right.) Then we can verify a property like "if there are finitely many processes in the system and some philosopher eats infinitely often then all philosophers eat infinitely often". This property holds under process-fairness, as philosophers release both their forks after eating.

▶ **Definition 4** (*DLSS verification problem*)**.** *Given a DLSS $\mathcal{S}$ and a regular objective $\mathcal{B}$ decide if there is a process-fair run of $\mathcal{S}$ whose limit configuration τ satisfies $\mathcal{B}$.*

Without any further restrictions we show that our problem is undecidable:

▶ **Theorem 5.** *The DLSS verification problem is undecidable. The result holds even if the DLSS is finite-state and every process uses at most 4 locks.*

This result is obtained by creating an unbounded chain of processes simulating a Turing machine. Each process memorizes the content of a position on the tape, and communicates with its neighbours by interleaving lock acquisitions. The trick for processes to exchange information by interleaving lock acquisitions was already used in [16], and requires a non-nested usage of locks.

The situation improves significantly if we assume nested usage of locks.

▶ **Definition 6.** *A process $\mathcal{A}_p$ is nested if it takes and releases locks according to a stack discipline, i.e., for all $x, y \in Var(p)$, for all paths $s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n$ in $\mathcal{A}_p$, with $op(a_1) = \mathtt{get}_x$, $op(a_n) = \mathtt{rel}_x$, $op(a_m) \neq \mathtt{rel}_x$ for all $m < n$: if $op(a_i) = \mathtt{get}_y$ for some $i < n$ then there exists $i < k < n$ such that $op(a_k) = \mathtt{rel}_y$. A DLSS is nested if all its processes are nested.*

We can state the first main result of the paper. Its proof is outlined in the next two sections.

▶ **Theorem 7.** *The DLSS verification problem for nested DLSS is* EXPTIME*-complete. It is in* PTIME *when the number of priorities in the specification automaton, and the maximal arity of processes are fixed.*

We can extend this result to DLSS where transition systems of each process are given by a pushdown automaton (see definitions in Section 5). The complexity remains the same as for finite state processes.

▶ **Theorem 8.** *The DLSS verification problem for nested pushdown DLSS is* EXPTIME*-complete. It is in* PTIME *when the number of priorities in the specification automaton, and the maximal arity of processes is fixed.*

3 Characterizing limit configurations

A configuration is a labeled tree. We give a characterization of such trees that are limit configurations of a process-fair run of a given DLSS. In the following section we will show that the set of limit configurations of a given DLSS is a regular tree language, which will imply the decidability of our verification problem.

▶ **Definition 9.** *Given a configuration τ with nodes ν, ν' and variables $x \in Var(\nu)$, $x' \in Var(\nu')$, we write $x \sim x'$ if $L(\nu)(x) = L(\nu')(x')$, so if x and x' are mapped to the same lock. The scope of a lock ℓ is the set $\{\nu : \ell \in L(\nu)(Var(\nu))\}$.*

▶ Remark 10. It is easy to see that in any configuration, the scope of a lock is a subtree.

We say that a node ν is labeled by an *unmatched* $\mathtt{get}$ if it is labeled by some $\mathtt{get}_x$ and there is no $\mathtt{rel}_x$ operation in the leftmost path starting from ν. Recall that $H(\tau)$ is the set of locks ℓ for which there is some node ν with an unmatched $\mathtt{get}_x$ and $L(\nu)(x) = \ell$.

We define a relation $\prec_H$ on $H(\tau)$ by letting $\ell \prec_H \ell'$ if there exist two nodes ν, ν' such that ν is an ancestor of ν', ν is labeled with an unmatched $\mathtt{get}$ of ℓ, and ν' is labeled with a $\mathtt{get}$ of ℓ'.

After these preparations we can state a central lemma giving a structural characterization of limit configurations of process-fair runs.

▶ **Lemma 11.** *A tree τ is the limit configuration of a process-fair run of a nested DLSS $\mathcal{S}$ if and only if*

F1 *The node labels in τ match the local transitions of $\mathcal{S}$.*

F2 *For every leaf ν every possible transition from $s(\nu)$ has operation $\mathtt{get}_x$ for some x with $L(\nu)(x) \in H(\tau)$.*

F3 *For every lock $\ell \in H(\tau)$ there are finitely many nodes with operations on ℓ, and there is a unique node labeled with an unmatched `get` of ℓ.*

F4 *The relation $\prec_H$ is acyclic.*

F5 *The relation $\prec_H$ has no infinite descending chain.*

Before presenting the proof of the previous lemma note that the main difficulty is the fact that some locks can be taken and never released. If $H(\tau) = \emptyset$ then from τ we can easily construct a run with limit configuration τ by exploiting the nested lock usage. This is because any local run can be executed from a configuration where all locks are available.

Proof. We start with the left-to-right implication. Suppose that we have a process-fair run $\tau_0 \xrightarrow{\nu_1,a_1} \tau_1 \xrightarrow{\nu_2,a_2} \cdots$ with limit configuration τ.

With every lock $\ell \in H(\tau)$ we associate the maximal position $m = m_\ell$ such that $op(a_m) = \mathtt{get}_x$ and $L(\nu_m)(x) = \ell$, so the position m_ℓ where ℓ is acquired for the last time (and never released after).

It remains to check the conditions of the lemma. The first one holds by definition of a run. The second condition is due to process fairness and soundness, since a process can always execute transitions other than acquiring a lock, and locks not in $H(\tau)$ are free infinitely often. All actions involving $\ell \in H(\tau)$ must happen before position m_ℓ, hence there are finitely many of them. Moreover, a lock cannot be acquired and never released more than once. This shows condition F3. Conditions F4 and F5 are both satisfied because if $\ell \prec_H \ell'$ then $m_\ell < m_{\ell'}$. Thus $\prec_H$ is acyclic and it cannot have infinite descending chains.

For the right-to-left implication, let τ satisfy all conditions of the lemma. In order to construct a run from τ we first build a total order $<$ on $H(\tau)$ that extends $\prec_H$ and has no infinite descending chain. Let $\ell'_0, \ell'_1, \ldots$ be some arbitrary enumeration of $H(\tau)$ (which exists as τ is countable, thus so is $H(\tau)$). For all i let $\downarrow \ell'_i = \{\ell' \in H(\tau) \mid \ell' \prec_H^+ \ell'_i\}$. As τ satisfies condition F3, the set of nodes that are ancestors of a node with an operation on ℓ'_i is finite. Since additionally by condition F5 there are no infinite descending chains for $\prec_H$, the set $\downarrow \ell'_i$ is finite as well (by König's lemma). As $\prec_H$ is acyclic by condition F4, we can chose some strict total order $<_i$ on $\downarrow \ell'_i$ that extends $\prec_H$. We define for all $\ell \in H(\tau)$ the index $m_\ell = \min\{i \in \mathbb{N} \mid \ell \in \downarrow \ell'_i\}$. Finally, we set $\ell < \ell'$ if either $m_\ell < m_{\ell'}$ or if $m_\ell = m_{\ell'}$ and $\ell <_{m_\ell} \ell'$. By definition $<$ is a strict total order on $H(\tau)$ with no infinite descending chains. Moreover it is easy to see that if $\ell \prec_H \ell'$ then $\ell < \ell'$. This is the case because $\ell \prec_H \ell'$ and $\ell' \prec_H^+ \ell_i$ implies $\ell \prec_H^+ \ell_i$, so $m_\ell \leq m_{\ell'}$.

Using the order $<$ on $H(\tau)$ we construct a process-fair run $\tau_0 \xrightarrow{+} \tau_1 \xrightarrow{+} \cdots$ with τ as limit configuration. During the construction we maintain the following invariant for every i:

> There exists $k_i \in \mathbb{N}$ such that all operations on locks ℓ_j with $j < k_i$ are already executed in τ_i (there is no operation on these locks in $\tau \setminus \tau_i$). Moreover, all other locks are free after executing τ_i: $H_i := H(\tau_i) = \{\ell_0, \ldots, \ell_{k_i-1}\}$.

For $i = 0$ the invariant is clearly satisfied as all locks are free ($k_0 = 0$).

For $i > 0$ we assume that there is a run $\tau_0 \xrightarrow{+} \tau_i$ and τ_i satisfies the invariant. Thus, all locks ℓ_j with $j < k_i$ are ultimately held and all other locks are free in τ_i.

We say that a leaf ν of τ_i is *available* if one of the following holds:

1. either there is a descendant $\nu' \neq \nu$ on the leftmost path from ν in τ with $H(\nu') = H(\nu)$ in τ,
2. or the left child ν' of ν in τ is labeled with an unmatched `get` of ℓ_{k_i}, and there is no further operation on ℓ_{k_i} in $\tau \setminus \tau_i$.

In particular, leaves of τ cannot be available. The strategy is to find the smallest available node ν in BFS order, and execute the actions on the left path from ν to ν'. The execution is possible as on this path there are no actions using locks from H_i and all other locks are free. Let τ_{i+1} denote the configuration thus obtained from τ_i. The invariant is satisfied after this execution, with $H_{i+1} = H_i$ in the first case above, resp. $H_{i+1} = H_i \cup \{\ell_{k_i}\}$ in the second case.

It remains to show that if a node is a leaf in τ_i for all i after some point, then it is a leaf in τ. This shows, in particular, that there always exists some available node.

Suppose that ν and i_0 are such that ν is a leaf of τ_i for all $i \geq i_0$. If ν becomes available at some point then it stays available in all future configurations, and there are finitely many nodes before ν in the BFS order. Thus ν cannot be available in some τ_i, as otherwise it would eventually be taken. Note that by the invariant (and soundness), no leaf of τ_i has the left child labeled by some `rel` operation. Moreover, every leaf ν of τ_i with left child ν' in τ labeled by `nop`, `spawn`(), or by some matched `get`, is available (the latter because we consider nested DLSS). Hence, the left child of ν must be labeled with an unmatched `get` of some $\ell \in H(\tau)$. Thus there is some unmatched `get` on a lock of $H(\tau)$ that is never executed.

Let m be the minimal index in the enumeration of $H(\tau)$ such that an unmatched `get` of ℓ_m in τ is never executed. By minimality of m, there exists i_1 such that $m = k_i$ for all $i \geq i_1$. After i_1, all operations on locks $\ell < \ell_m$ have been executed. Thus, as $<$ extends $\prec_H$, all unmatched `get` operations that have some descendant in τ with operation on ℓ_m, have been executed. By the previous argument, the nodes with left child not labeled with an unmatched `get` cannot stay leaves forever. Hence, all nodes whose left child has some operation on ℓ_m eventually become leaves. The ones with matched `get` or other operations are then available and eventually executed.

Hence, after some point the only remaining operations on ℓ_m are unmatched `get`. Furthermore by the condition F3 of the lemma there is exactly one. As a result, when it is reached and all other operations on ℓ_m have been executed, it becomes available, and is thus eventually executed, contradicting the definition of m.

This proves that the limit of the run we have constructed is τ. Observe finally that the run is process-fair because of condition F2 of the lemma. ◀

The next lemma is an important step in the proof as it simplifies condition F4 of Lemma 11. This condition talks about the existence of a global order on some locks. The next lemma replaces this order with local orders in each of the nodes. These orders can be guessed by a finite automaton.

▶ **Lemma 12.** *Suppose that τ satisfies the first three conditions of Lemma 11. The relation $\prec_H$ is acyclic if and only if there is a family of strict total orders $<_\nu$ over a subset of variables from $Var(\nu)$such that:*

F4.1 *x is ordered by $<_\nu$ if and only if $L(\nu)(x) \in H(\tau)$.*

F4.2 *if $x <_\nu x'$, ν' is a child of ν, and $y, y' \in Var(\nu')$ are such that $x \sim y$ and $x' \sim y'$ then $y <_{\nu'} y'$.*

F4.3 *if $x, x' \in Var(\nu)$ and $L(\nu)(x) \prec_H L(\nu)(x')$ then $x <_\nu x'$.*

4 Recognizing limit configurations

Recall that a configuration is a possibly infinite tree with five labels p, s, a, L, H. As we have mentioned in Remark 2, configurations need actually only three labels p, s, a. The other two can be calculated from the tree. Hence, configurations are labeled trees with node labels coming from a finite alphabet. Our goal in this section is to define a tree automaton recognizing limit configurations of process-fair runs of a given DLSS.

Our plan is to check the conditions (F1-5) of Lemma 11. Actually we will check (F1-3,5) and the conditions of Lemma 12 that are equivalent to F4 of Lemma 11.

We first observe that since our processes are finite state it is immediate to construct a nondeterministic tree automaton $\mathcal{B}_1$ verifying condition F1. This automaton just verifies local constraints between the labeling of a node and the labelings of its children. The constraints talk only about the labels p, s, a. The automaton does not need any acceptance condition, every run is accepting. We will say τ is *process-consistent* if it is accepted by $\mathcal{B}_1$.

Checking condition F2 is more complicated because it refers to a set $H(\tau)$ of locks that are ultimately held by some process. Our approach will be to define four types of predicates and color the nodes of τ with these predicates. From a correct coloring of τ it will be easy to read out $H(\tau)$. Then we will show that the correct coloring can be characterized by conditions verifiable by Büchi tree automata. The coloring will be also instrumental in checking the remaining conditions F3, F4, F5.

For a configuration τ, a node ν and a variable $x \in \mathit{Var}(\nu)$ we define four predicates.

- $\nu \models \mathit{keeps}(x)$ if at ν process $p(\nu)$ holds the lock $\ell = L(\nu)(x)$ and never releases it: $\ell \in H(\nu')$ for every left descendant ν' of ν.
- $\nu \models \mathit{ev\text{-}keeps}(x)$ if $\nu \not\models \mathit{keeps}(x)$ and there is a descendant ν' of ν and a variable $x' \in \mathit{Var}(\nu')$ with $x \sim x'$ and $\nu' \models \mathit{keeps}(x')$.
- $\nu \models \mathit{avoids}(x)$ if neither $p(\nu)$ nor any descendant takes $\ell = L(\nu)(x)$, namely $\ell \notin H(\nu')$ for every descendant ν' of ν (including ν).
- $\nu \models \mathit{ev\text{-}avoids}(x)$ if $\nu \not\models \mathit{avoids}(x)$ and on every path from ν there is ν' such that $\nu' \models \mathit{avoids}(x)$.

Observe a different quantification used in *ev-keeps* and *ev-avoids*. In the first case we require one ν' to exist, in the second we want that such a ν' exists on every path.

The next lemma shows how we can use the coloring to determine $H(\tau)$.

▶ **Lemma 13.** *Let τ be a process-consistent configuration. A lock $\ell \in H(\tau)$ if and only if there is a node ν of τ and a variable $x \in \mathit{Var}(\nu)$ such that $\nu \models \mathit{keeps}(x)$ and $L(\nu)(x) = \ell$.*

Proof. Follows from the definitions, since $\nu \models \mathit{keeps}(x)$ if and only if $\ell \in H(\nu')$ for every left descendant ν' of ν. ◀

The above conditions define a *semantically correct* coloring of nodes of a configuration τ by sets of predicates

$$\mathcal{C}(\nu) = \{P(x) : x \in \mathit{Var}(\nu), \nu \models P(x)\}$$

where $P(x)$ is one of $\mathit{keeps}(x), \mathit{ev\text{-}keeps}(x), \mathit{avoids}(x), \mathit{ev\text{-}avoids}(x)$. Observe that the four predicates are mutually exclusive, but it may be also the case that none of them holds. We say that a variable $x \in \mathit{Var}(\nu)$ is *uncolored* in ν if $\mathcal{C}(\nu)$ contains no predicate on x.

We now describe consistency conditions on a coloring of configurations guaranteeing that a coloring is semantically correct.

Before moving forward we introduce one piece of notation. A node that is a right child, namely a node of a form $\nu 1$ is due to $\texttt{spawn}(q, \sigma)$ operation. More precisely $op(\nu 0) = \texttt{spawn}(q, \sigma)$. We refer to this σ as $\sigma(\nu 1)$.

A coloring of a configuration τ is *branch-consistent* if for every node ν of τ and every variable $x \in \mathit{Var}(\nu)$ the following conditions are satisfied.

- If ν has one successor $\nu 0$ then $\nu 0$ inherits the colors from ν except for two cases depending on $op(\nu 0)$, i.e, the operation used to obtain $\nu 0$:

  - If *ev-keeps*(x) is in $\mathcal{C}(\nu)$ and the operation is $\mathtt{get}_x$ then $\mathcal{C}(\nu 0)$ must have either *ev-keeps*(x) or *keeps*(x).
  - If *ev-avoids*(x) is in $\mathcal{C}(\nu)$ and the operation is $\mathtt{rel}_x$ then $\mathcal{C}(\nu 0)$ must have either *ev-avoids*(x) or *avoids*(x).
- If ν has two successors $\nu 0$, $\nu 1$, and there is no y with $\sigma(\nu 1)(y) = x$ then $\nu 0$ inherits x color from ν and there is no constraint due to x on colors in $\nu 1$.
- If ν has two successors and $x = \sigma(\nu_1)(y)$ for some $y \in Var(\nu 1)$ then
  - If *keeps*(x) in $\mathcal{C}(\nu)$ then *keeps*(x) in $\mathcal{C}(\nu 0)$ and *avoids*(y) in $\mathcal{C}(\nu 1)$.
  - If *avoids*(x) in $\mathcal{C}(\nu)$ then *avoids*(x) in $\mathcal{C}(\nu 0)$ and *avoids*(y) in $\mathcal{C}(\nu 1)$.
  - If *ev-keeps*(x) in $\mathcal{C}(\nu)$ then either
    - *ev-keeps*(x) in $\mathcal{C}(\nu 0)$ and either *avoids*(y) or *ev-avoids*(y) in $\nu 1$, or
    - *ev-keeps*(y) in $\mathcal{C}(\nu 1)$ and either *avoids*(x) or *ev-avoids*(x) in $\nu 0$.
  - If *ev-avoids*(x) is ν then *ev-avoids*(x) in $\mathcal{C}(\nu 0)$ and *ev-avoids*(y) in $\mathcal{C}(\nu 1)$.

Next we describe when a coloring is *eventuality-consistent*. An *ev-trace* is a sequence of pairs $(\nu_1, x_1), (\nu_2, x_2), \dots$ where :

- $\nu_1, \nu_2, \dots$ is a path in τ,
- $x_i \in Var(\nu_i)$; moreover $x_{i+1} = x_i$ if ν_{i+1} is the left successor of ν_i, and $\sigma(\nu_{i+1})(x_{i+1}) = x_i$ if ν_{i+1} is the right successor of ν_i.
- *ev-keeps*(x_i) or *ev-avoids*(x_i) is in $\mathcal{C}(\nu_i)$.

Observe that it follows that it cannot be the case that we have *ev-keeps*(x_i) and *ev-avoids*(x_{i+1}) or vice versa. A coloring is eventuality-consistent if every ev-trace in the coloring of a configuration is finite.

Finally, a coloring is *recurrence-consistent* if for every ν and uncolored $x \in Var(\nu)$ the lock $\ell = L(\nu)(x)$ is taken and released infinitely often below ν.

A coloring is *syntactically correct* if it is branch-consistent, eventuality-consistent, and recurrence-consistent. We show that syntactically correct colorings characterize semantically correct colorings. The two implications are stated separately as the statements are slightly different.

▶ **Lemma 14.** *If τ is a limit configuration and $\mathcal{C}$ is a semantically correct coloring of τ then $\mathcal{C}$ is syntactically correct.*

For the other direction, we prove a more general statement without assuming that τ is a limit configuration. This is important as ultimately we will use the consistency properties to test if τ is a limit configuration.

▶ **Lemma 15.** *If τ is a configuration and $\mathcal{C}$ a syntactically correct coloring of τ, then $\mathcal{C}$ is semantically correct.*

Having a correct coloring will help us to verify all conditions of Lemma 11. Condition F2 refers to $L(\nu)(x) \in H(\tau)$. We need another labeling to be able to express this.

A *syntactic H-labeling* of τ assigns to every node ν a subset $H^s(\nu) \subseteq Var(\nu)$. We require the following properties:

- For the root ε we have $H^s(\varepsilon) = \emptyset$.
- If $\nu 0$ exists: $x \in H^s(\nu 0)$ if and only if $x \in H^s(\nu)$.
- If $\nu 1$ exists: $y \in H^s(\nu 1)$ if and only if either $\sigma(\nu 1)(y) = \mathtt{new}$ and $\nu 1 \models$ *ev-keeps*(y), or $\sigma(\nu 1)(y) = x$ and $\nu \models$ *ev-keeps*(x).

It is clear that every configuration tree has a unique H^s labelling.

▶ **Lemma 16.** *Let τ be a process-consistent configuration with syntactically correct coloring. For every node ν and variable $x \in \mathit{Var}(\nu)$ we have: $L(\nu)(x) \in H(\tau)$ if and only if $x \in H^s(\nu)$.*

Thanks to Lemma 16 we obtain

▶ **Lemma 17.** *Let τ be a process-consistent configuration with a syntactically correct coloring. Condition F2 of Lemma 11 holds for τ if and only if for every leaf ν of τ, every possible transition from $s(\nu)$ has some $\mathtt{get}_x$ operation with $x \in H^s(\nu)$.*

▶ **Lemma 18.** *Let τ be a process-consistent configuration with a syntactically correct coloring. Then condition F3 of Lemma 11 holds for τ.*

It remains to deal with conditions F4 and F5 of Lemma 11. Condition F4 is more difficult to check as it requires to find an acyclic relation with some properties. Fortunately Lemma 12 gives an equivalent condition talking about a family of local orders $<_\nu$ for every node ν of a configuration. An automaton can easily guess such a family of orders. We show that it can also check the required properties.

A *consistent order labeling* assigns to every node ν of τ a total order $<_\nu$ on some subset of $\mathit{Var}(\nu)$. The assignment must satisfy the following conditions for every node ν:

1. x is ordered by $<_\nu$ if and only if $x \in H^s(\nu)$,
2. if $x <_\nu x'$ and $x, x' \in \mathit{Var}(\nu 0)$ then $x <_{\nu 0} x'$,
3. if $x <_\nu x'$, $\nu 1$ exists, and $\sigma(\nu 1)(y) = x$, $\sigma(\nu 1)(y') = x'$ then $y <_{\nu 1} y'$,
4. if $\nu \models \mathit{keeps}(x)$ and $y <_\nu x$ then $\nu \models \mathit{keeps}(y)$ or $\nu \models \mathit{avoids}(y)$.

▶ **Lemma 19.** *Let τ be a process-consistent configuration with a syntactically correct coloring. A family of local orders $<_\nu$ is a consistent order labeling of τ if and only if it satisfies the conditions of Lemma 12.*

We consider now condition F5. We say that a consistent order labeling of τ admits an *infinite descending chain* if there exist a sequence of nodes $\nu_1, \nu_2, \dots$ and variables $(x_i)_i, (y_i)_i$ such that for every $i > 0$: (i) ν_i is an ancestor of ν_{i+1}, (ii) $y_i \sim x_{i+1}$, and (iii) $y_i <_{\nu_i} x_i$.

▶ **Lemma 20.** *Let τ be a process-consistent configuration with a syntactically correct coloring. If $\prec_H$ has no infinite descending chain then there is a consistent order labeling of τ with no infinite descending chain. If $\prec_H$ has an infinite descending chain then every consistent order labeling of τ admits an infinite descending chain.*

The next proposition summarizes the development of this section stating that all the relevant properties can be checked by a Büchi tree automaton.

▶ **Proposition 21.** *For a given DLSS, there is a non-deterministic Büchi tree automaton $\widehat{\mathcal{B}}$ accepting exactly the limit configurations of process-fair runs of DLSS. The size of $\widehat{\mathcal{B}}$ is linear in the size of the DLSS and exponential in the maximal arity of the DLSS.*

We will show that the previous proposition yields an Exptime algorithm. We match it with an Exptime lower bound to obtain completeness.

▶ **Proposition 22.** *The DLSS verification problem for nested DLSS and Büchi objective is Exptime-hard. The result holds even if the Büchi objective refers to a single process.*

The hardness proof involves a reduction from the problem of determining whether the intersection of the languages of k deterministic tree automata over binary trees is empty. To achieve this, we create a DLSS that simulates all the tree automata concurrently. Each node of the tree in the intersection is simulated by a process, which encodes a state for each automaton through the locks it holds. So each process creates two children with whom it shares locks. The children are able to access the states of the parent by the following technique: Suppose processes p and q share locks 0 and 1, and p acquires one lock and retains it indefinitely. In this scenario, q can guess the lock chosen by p and try to acquire the other lock. If q guesses incorrectly, the system deadlocks. However, if the guess is correct, the execution continues, and q knows about the lock held by p.

Now we have all ingredients for the proof of Theorem 7:

Proof of Theorem 7. The lower bound follows from Proposition 22.

For the upper bound we use the Büchi tree automaton $\hat{\mathcal{B}}$ recognizing limit configurations of the DLSS (Proposition 21).

We build the product of $\hat{\mathcal{B}}$ with the regular objective automaton $\mathcal{A}$, which is a parity tree automaton. From $\hat{\mathcal{B}} \times \mathcal{A}$ we can obtain with a bit more work an equivalent parity tree automaton $\mathcal{C}$ with the same number of priorities, plus one. For this we modify the rank function in order to only store in the state the maximal priority seen between two consecutive occurrences of Büchi accepting states, and make the maximal priority visible at the next Büchi state. When the state of the $\hat{\mathcal{B}}$ component is not a Büchi state, the priority is odd and lower than all the ones of $\mathcal{A}$.

By Proposition 21, $\mathcal{C}$ is non-empty if and only if there exists a limit configuration of the system that satisfies the regular objective $\mathcal{A}$. Moreover, we know that $\hat{\mathcal{B}}$ has size linear in the size of the DLSS and exponential only in the maximal arity of processes. So $\mathcal{C}$ has size that is exponential w.r.t. the DLSS and the objective, and polynomial size if the maximal arity is fixed.

Finally, non-emptiness of $\mathcal{C}$ amounts to solve a parity game of the same size as $\mathcal{C}$: player Automaton chooses transitions of $\mathcal{C}$, and player Pathfinder chooses the direction (left/right child). To sum up, we obtain a parity game of exponential size, so solving the game takes exponential time since the number of priorities is polynomial. If both the number of priorities and the maximal arity are fixed, the game can be solved in polynomial time. ◀

5 Pushdown systems with locks

Till now every process has been a finite state system. Here we consider the case when processes can be pushdown automata. The definition of a *pushdown DLSS* is the same as before but now each automaton $\mathcal{A}_p$ is a deterministic pushdown automaton.

We will reduce our verification problem to the emptiness test of a nondeterministic pushdown automata on infinite trees. These automata will have parity acceptance conditions. While in general testing emptiness of such automata is EXPTIME-complete, we will notice that the automata we construct have a special form allowing to test emptiness in PTIME for a fixed number of ranks in the parity condition.

We start by defining *pushdown tree automata*. We work with a ranked alphabet $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, so a letter determines whether a node has zero, one or two children. Our automaton will be quite standard but for an additional stack instruction. Apart standard `pop` and `push`(a), we have a `reset` instruction that empties the stack. A pushdown tree automaton is a tuple $(Q, \Sigma, \Gamma, q^0, \perp, \delta, \Omega)$, where Q is a finite set of states, Σ an input alphabet, Γ a stack alphabet, $q^0 \in Q$ an initial state, $\perp \in \Gamma$ a bottom stack symbol, and

$\Omega : Q \to \{1, \dots, d\}$ a parity condition. Finally, δ is a partial transition function taking as the arguments the current state q, the current input letter a, and the current stack symbol γ. The form of transitions in δ depends on the rank of the letter a:

- For $a \in \Sigma_0$, we have $\delta(q, a, \gamma) = \top$ for a special symbol $\top$. This means that the automaton accepts in a leaf of the tree if δ is defined.
- For $a \in \Sigma_1$, we have $\delta(q, a, \gamma) = (q', \mathtt{instr})$ where instr is one of the stack instructions.
- For $a \in \Sigma_2$, we have $\delta(q, a, \gamma) = ((q_l, \mathtt{instr}_l), (q_r, \mathtt{instr}_r))$, so now we have two states, going to the left and right, respectively, and two separate stack instructions.

A run of such an automaton on a Σ-labeled tree is an assignment of configurations to nodes of the tree; each configuration has the form (q, w) where $q \in Q$ is a state and $w \in \Gamma^+$ is a sequence of stack symbols representing the stack (top symbol being the leftmost). The root is labeled with $(q^0, \perp)$. The labelling of children must depend on the labeling of the parent according to the transition function δ. In particular, if a leaf of the tree is labeled a and has assigned a configuration (q, w) then $\delta(q, a, \gamma)$ must be defined, where γ is the leftmost symbol of w. A run is accepting if for every infinite path the sequence of assigned states satisfies the max parity condition given by Ω: the maximum of ranks of states seen on the path must be even.

We say that a pushdown tree automaton is *right-resetting* if for every transition $\delta(q, a, \gamma) = ((q_l, \mathtt{instr}_l), (q_r, \mathtt{instr}_r))$ we have that $\mathtt{instr}_r$ is reset.

▶ **Proposition 23.** *For a fixed d, the emptiness problem for right-resetting pushdown tree automata with a parity condition over ranks $\{1, \dots, d\}$ can be solved in* Ptime.

Proof. We consider the representative case of $d = 3$. Suppose we are given a right-resetting pushdown tree automaton $\mathcal{A} = (Q, \Sigma, \Gamma, q^0, \perp, \delta, \Omega)$.

The first step is to construct a pushdown word automaton $\mathcal{A}^l(G_1, G_2, G_3)$ depending on three sets of states $G_1, G_2, G_3 \subseteq Q$. The idea is that $\mathcal{A}^l$ simulates the run of $\mathcal{A}$ on the leftmost branch of a tree. When $\mathcal{A}$ has a transition going both to the left and to the right then $\mathcal{A}^l$ goes to the left and checks if the state going to the right is in an appropriate G_i. This means that $\mathcal{A}^l$ works over the alphabet Σ^l that is the same as Σ but all letters from Σ_2 have rank 1 instead of 2. The states of $\mathcal{A}^l(G_1, G_2, G_3)$ are $Q \times \{1, 2, 3\}$ with the second component storing the maximal rank of a state seen so far on the run. The transitions of $\mathcal{A}^l(G_1, G_2, G_3)$ are defined according to the above description. We make precise only the case for a transition of $\mathcal{A}$ of the form $\delta(q, a, \gamma) = ((q_l, \mathtt{instr}_l), (q_r, \mathtt{instr}_r))$. In this case, $\mathcal{A}^l$ has a transition $\delta^l((q, i), a, \gamma) = ((q_l, \max(i, \Omega(q_l))), \mathtt{instr}_l)$ if $q_r \in G_{\max(i, \Omega(q_r))}$. Observe that $\mathtt{instr}_r$ is necessarily reset as $\mathcal{A}$ is right-resetting.

The next step is to observe that for given sets G_1, G_2, G_3 we can calculate in Ptime the set of states from which $\mathcal{A}^l(G_1, G_2, G_3)$ has an accepting run.

The last step is to compute the fixpoint expression below in the lattice of subsets of Q. What the fixpoint computation does can be described at high-level as follows. While the word pushdown automaton $\mathcal{A}^l$ takes care of the parity condition on tree paths that are ultimately left paths, the sets G_i do this for paths that branch to the right infinitely often. For such paths we need for example to guarantee through set G_3 that priority 3 is seen finitely often. We do this through a least fixpoint computation for G_3. For G_2 we compute a greatest fixpoint since we want priority 2 to be seen infinitely often. Finally, for G_1 we compute a least fixpoint since priority 1 should be seen finitely often before seeing priority 2:

$$W = \mathsf{LFP}X_3.\ \mathsf{GFP}X_2.\ \mathsf{LFP}X_1.\ P(X_1, X_2, X_3) \qquad \text{where}$$
$$P(X_1, X_2, X_3) = \{q : \mathcal{A}^l(X_1, X_2, X_3) \text{ has an accepting run from } q\}\ .$$

Observe that $P : \mathcal{P}(Q)^3 \to \mathcal{P}(Q)$ is a monotone function over the lattice of subsets of Q. Computing W requires at most $|Q|^3$ computations of P for different triples of sets of states.

We claim that $\mathcal{A}$ has an accepting run from a state q, if and only if, $q \in W$. The proof can be found in the full version. ◀

Proof of Theorem 8. The lower bound follows already from Theorem 7.

For the upper bound we reuse the Büchi tree automaton $\hat{\mathcal{B}}$ from Proposition 21. This time $\hat{\mathcal{B}}$ is a pushdown tree automaton, however it is right-resetting because processes are spawned with empty stack. We follow the lines of the proof of Theorem 7, building the product of $\hat{\mathcal{B}}$ with the regular objective automaton $\mathcal{A}$, and constructing an equivalent parity, right-resetting pushdown tree automaton $\mathcal{C}$. Proposition 23 concludes the proof. ◀

6 Conclusions

We have considered verification of parametric lock sharing systems where processes can spawn other processes and create new locks. Representing configurations as trees, and the notion of the limit configuration, are instrumental in our approach. We believe that we have made stimulating observations about this representation. It is very easy to express fairness as a property of a limit configuration. Many interesting properties, including liveness, can be formulated very naturally as properties of limit trees (cf. page 6). Moreover, there are structural conditions characterizing when a tree is a limit configuration of a run of a given system (Lemma 12).

We expect that the parameters in Theorem 8 will be usually quite small. As the dining philosophers example suggests, for many systems the maximal arity should be quite small (cf. Figure 1). Indeed, the maximal arity of the system corresponds to the tree width of the graph where process instances are nodes and edges represent sharing a lock. The maximal priority will be often 3. In our opinion, most interesting properties would have the form "there is a left path such that" or "all left paths are such that", and these properties need only automata with three priorities. So in this case our verification algorithm is in PTIME.

Our handling of pushdown processes is different from the literature. Most of our development is done for finite state processes, while the transition to pushdown process is handled through right-resetting concept. Proposition 23 implies that in our context pushdown processes are essentially as easy to handle as finite processes.

As further work it would be interesting to see if it is possible to extend our approach to treat join operation [12]. An important question is how to extend the model with some shared state and still retain decidability for the pushdown case.

References

1 Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 685–725. Springer, 2018. doi:10.1007/978-3-319-10575-8_21.

2 S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Sparsa Roychowdhury. Revisiting underapproximate reachability for multipushdown systems. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 387–404. Springer, 2020. doi:10.1007/978-3-030-45190-5_21.

3 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. The complexity of bounded context switching with dynamic thread creation. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 111:1–111:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.111.

4 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of thread pools. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:10.1145/3498678.

5 Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015. doi:10.2200/S00658ED1V01Y201508DCT013.

6 Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR'97: Concurrency Theory, 8th International Conference, Warsaw*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. doi:10.1007/3-540-63141-0_10.

7 Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 473–487. Springer, 2005. doi:10.1007/11539452_36.

8 Xvisor commit message fixing issue:. URL: https://github.com/xvisor/xvisor/commit/e5dd8291b5e3f0c552b9aacc73ef2f000ae14c09.

9 Marcio Diaz and Tayssir Touili. Dealing with priorities and locks for concurrent programs. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis – 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2017. doi:10.1007/978-3-319-68167-2_15.

10 Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. *J. ACM*, 63(1):10:1–10:48, 2016. doi:10.1145/2842603.

11 Marie Fortin, Anca Muscholl, and Igor Walukiewicz. Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification – 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 155–175. Springer, 2017. doi:10.1007/978-3-319-63390-9_9.

12 Thomas Martin Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation – 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2011. doi:10.1007/978-3-642-18275-4_15.

13 S. A. German and P. A. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

14 Matthew Hague. Parameterised pushdown systems with non-atomic writes. In Supratik Chakraborty and Amit Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, volume 13 of *LIPIcs*, pages 457–468. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.FSTTCS.2011.457.

15 Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 27–36, 2009. doi:10.1109/LICS.2009.45.

16 Vineet Kahlon, Franjo Ivancić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 505–518, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11513988_49.

17 Sebastian Kenter. *Lock-sensitive reachability analysis for parallel recursive programs with dynamic creation of threads and locks: a graph-based approach.* PhD thesis, University of Münster, Germany, 2022. URL: https://nbn-resolving.org/urn:nbn:de:hbz:6-21089543742.

18 Peter Lammich. *Lock sensitive analysis of parallel programs.* PhD thesis, University of Münster, 2011. URL: https://nbn-resolving.org/urn:nbn:de:hbz:6-43459441169.

19 Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Contextual locking for dynamic pushdown networks. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis – 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 477–498. Springer, 2013. doi:10.1007/978-3-642-38856-9_25.

20 Peter Lammich, Markus Müller-Olm, and Alexander Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 – July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 525–539. Springer, 2009. doi:10.1007/978-3-642-02658-4_39.

21 Anca Muscholl, Helmut Seidl, and Igor Walukiewicz. Reachability for dynamic parametric processes. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation – 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2017. doi:10.1007/978-3-319-52234-0_23.

22 Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005. doi:10.1007/978-3-540-31980-1_7.

23 Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 161–170. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.9.

24 Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. Reachability of scope-bounded multistack pushdown systems. *Inf. Comput.*, 275:104588, 2020. doi:10.1016/j.ic.2020.104588.

25 Kazuhide Yasukata, Takeshi Tsukada, and Naoki Kobayashi. Verification of higher-order concurrent programs with dynamic resource creation. In Atsushi Igarashi, editor, *Programming Languages and Systems – 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 335–353, 2016. doi:10.1007/978-3-319-47958-3_18.

A General Approach to Under-Approximate Reasoning About Concurrent Programs

Azalea Raad ✉ 🏠
Imperial College London, UK

Julien Vanegue ✉
Bloomberg, New York, NY, USA

Josh Berdine ✉
Skiplabs, London, UK

Peter O'Hearn ✉
University College London, UK
Lacework, London, UK

Abstract

There is a large body of work on concurrent reasoning including Rely-Guarantee (RG) and Concurrent Separation Logics. These theories are *over-approximate*: a proof identifies a *superset* of program behaviours and thus implies the absence of certain bugs. However, failure to find a proof does not imply their presence (leading to *false positives* in over-approximate tools). We describe a general theory of *under-approximate* reasoning for concurrency. Our theory incorporates ideas from Concurrent Incorrectness Separation Logic and RG based on a subset rather than a superset of interleavings. A strong motivation of our work is detecting *software exploits*; we do this by developing *concurrent adversarial separation logic* (CASL), and use CASL to detect *information disclosure attacks* that uncover sensitive data (e.g. passwords) and *out-of-bounds attacks* that corrupt data. We also illustrate our approach with classic concurrency idioms that go beyond prior under-approximate theories which we believe can inform the design of future concurrent bug detection tools.

2012 ACM Subject Classification Theory of computation → Separation logic; Theory of computation → Programming logic; Theory of computation → Program analysis; Security and privacy → Logic and verification

Keywords and phrases Under-approximate reasoning, incorrectness logic, bug detection, software exploits, separation logic

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.25

Related Version *Extended Version*: `https://www.soundandcomplete.org/papers/CONCUR2023/CASL/appendix.pdf`

Funding *Azalea Raad*: UKRI Future Leaders Fellowship under grant number MR/V024299/1

1 Introduction

Incorrectness Logic (IL) [16] presents a formal foundation for proving the *presence* of bugs using *under-approximation*, i.e. focusing on a *subset* of behaviours to ensure one detects only *true positives* (real bugs) rather than *false positives* (spurious bug reports). This is in contrast to verification frameworks proving the *absence* of bugs using *over-approximation*, where a *superset* of behaviours is considered. The key advantage of under-approximation is that tools underpinned by it are accompanied by a *no-false-positives* (NFP) theorem *for free*, ensuring all bugs reported are real bugs. This has culminated in a successful trend in automated static analysis tools that use under-approximation for bug detection, e.g. RacerD [3] for data race detection in Java programs, the work of Brotherston et al. [4] for deadlock detection, and Pulse-X [13] which uses the under-approximate theory of ISL (incorrectness separation logic, an IL extension) [17] for detecting memory safety bugs such as use-after-free errors. All three tools are currently industrially deployed and are state-of-the art techniques: RacerD significantly outperforms other race detectors in terms of bugs found and fixed, while Pulse-X has a higher fix-rate than the industrial Infer tool [7] used widely at Meta, Amazon and Microsoft. IL and ISL, though, only support bug detection in *sequential* programs.

We present *concurrent adversarial separation logic* (CASL, pronounced "castle"), a general, under-approximate framework for detecting concurrency bugs and exploits, including a hitherto unsupported class of bugs. Inspired by adversarial logic [22], we model a vulnerable program $\mathsf{C_v}$ and its attacker (adversarial) $\mathsf{C_a}$ as the concurrent program $\mathsf{C_a} \,||\, \mathsf{C_v}$, and use the compositional principles of CASL to detect vulnerabilities in $\mathsf{C_v}$. CASL is a *parametric* framework that can be instantiated for a range of bugs/exploits. CASL combines under-approximation with ideas from RGSep [20] and concurrent separation logic (CSL) [15] – we chose RGSep rather than rely-guarantee [11] for compositionality (see p. 7). However, CASL does not merely replace over- with under-approximation in RGSep/CSL: CASL includes an additional component witnessing (*under-approximating*) the interleavings leading to bugs.

CASL builds on *concurrent incorrectness separation logic* (CISL) [18]. However, while CISL was designed to capture the reasoning in cutting-edge tools such as RacerD, CASL explicitly goes beyond these tools. Put differently, CISL aspired to be a *specialised* theory of concurrent under-approximation, oriented to existing tools (and inheriting their limitations), whereas CASL aspires to be more *general.* In particular, in our private communication with CISL authors they have confirmed two key limitations of CISL. First, CISL can detect certain bugs compositionally only by encoding buggy executions as normal ones. While this is sufficient for bugs where encountering a bug does not force the program to terminate (e.g. data races), it cannot handle bugs with *short-circuiting semantics*, e.g. null pointer exceptions, where the execution is halted on encountering the bug (see §2 for details). Second and more significantly, CISL cannot *compositionally* detect a large class of bugs, *data-dependent* bugs, where a bug occurs only under certain interleavings and concurrent threads affect the control flow of one another. To see this, consider the program $\mathsf{P} \triangleq x := 1 \,||\, a := x; \mathsf{if}\ (a)\ \mathsf{error}$, where the left thread, τ_1, writes 1 to x, the right thread, τ_2, reads the value of x in a and subsequently errors if $a{\neq}0$. That is, the error occurs only in interleavings where τ_1 is executed before τ_2, and the two threads synchronise on the value of x; i.e. τ_1 affects the control flow of τ_2 and the error occurrence is *dependent* on the *data* exchange between the threads.

Such data-dependency is rather prevalent as threads often synchronise via *data exchange.* Moreover, a large number of security-breaking *software exploits* are data-dependent bugs. An exploit (or *attack*) is code that takes advantage of a bug in a vulnerable program to cause unintended or erroneous behaviours. *Vulnerabilities* are bugs that lead to critical security compromises (e.g. leaking secrets or elevating privileges). Distinguishing vulnerabilities from benign bugs is a growing problem; understanding the exploitability of bugs is a time-consuming process requiring expert involvement, and large software vendors rely on automated exploitability analysis to prioritise vulnerability fixing among a sheer number of bugs. Rectifying vulnerabilities in the field requires expensive software mitigations (e.g. addressing Meltdown [14]) and/or large-scale recalls. It is thus increasingly important to detect vulnerabilities pre-emptively during development to avoid costly patches and breaches.

To our knowledge, CASL is the *first* under-approximate theory that can detect *all* categories of concurrency bugs (including data-dependent ones) *compositionally* (by reasoning about each thread in isolation). CASL is strictly stronger than CISL and supports all CISL reasoning principles. Moreover, CASL is the *first* under-approximate and compositional theory for exploit detection. We instantiate CASL to detect *information disclosure attacks* that uncover sensitive data (e.g. Heartbleed [8]) and *out-of-bounds attacks* that corrupt data (e.g. zero allocation [21]). Thanks to CASL soundness, each CASL instance is automatically accompanied by an NFP theorem: all bugs/exploits identified by it are true positives.

Contributions and Outline. Our contributions (detailed in §2) are as follows. We present CASL (§3) and prove it sound, with the full proof given in the accompanying technical appendix [19]. We instantiate CASL to detect information disclosure attacks on stacks (§4) and heaps [19, §C] and memory safety attacks [19, §D]. We also develop an under-approximate analogue of RG that is simpler but less expressive than CASL [19, §E and §F]. We discuss related work in §5.

2 Overview

CISL and Its Limitations. CISL [18] is an under-approximate logic for detecting bugs in concurrent programs with a built-in *no-false-positives theorem* ensuring all bugs detected are true bugs. Specifically, CISL allows one to prove triples of the form $[p]\ \mathsf{C}\ [\epsilon : q]$, stating that *every* state in q is reachable by executing C starting in *some* state in p, under the (exit) condition ϵ that may be either *ok* for normal (non-erroneous) executions, or $\epsilon \in \textsc{ErExit}$ for erroneous executions, where ErExit contains erroneous conditions. The CISL authors identify *global* bugs as those that are due to the interaction between two or more concurrent threads and arise only under certain interleavings. To see this, consider the examples below [18], where we write τ_1 and τ_2 for the left and right threads in each example, respectively:

$$\textsc{l}\colon \mathsf{free}(x) \;\Big|\Big|\; \textsc{l}'\colon \mathsf{free}(x) \qquad (\textsc{DataAgn}) \qquad\qquad \begin{array}{l} \mathsf{free}(x); \\ [z] := 1; \end{array} \;\Bigg|\Bigg|\; \begin{array}{l} a := 0;\ a := [z]; \\ \mathsf{if}\ (a{=}1)\ \textsc{l}\colon [x] := 1 \end{array} \qquad (\textsc{DataDep})$$

In an interleaving of DataAgn in which τ_1 is executed after (resp. before) τ_2, a double-free bug is reached at L (resp. L′). Analogously, in a DataDep interleaving where τ_2 is executed after τ_1, value 1 is read from z in a, the condition of if is met and thus we reach a use-after-free bug at L. Raad et al. [18] categorise global bugs as either *data-agnostic* or *data-dependent*, denoting whether concurrent threads contributing to a global bug may affect the *control flow* of one another. For instance, the bug at L in DataDep is data-dependent as τ_1 may affect the control flow of τ_2: the value read in $a := [z]$, and subsequently the condition of if and whether $\textsc{l}\colon [x] := 1$ is executed depend on whether τ_2 executes $a := [z]$ before or after τ_1 executes $[z] := 1$. By contrast, the threads in DataAgn cannot affect the control flow of one another; hence the bugs at L and L′ are data-agnostic.

$$\textsc{CISL-Par}\quad \frac{[P_1]\,\mathsf{C}_1\,[ok\!:\!Q_1] \qquad [P_2]\,\mathsf{C}_2\,[ok\!:\!Q_2]}{[P_1 * P_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [ok\!:\ Q_1 * Q_2]}$$

In *certain cases*, CISL can detect data-agnostic bugs compositionally (i.e. by analysing each thread in isolation) by encoding buggy executions as normal (*ok*) ones and then using the CISL-Par rule shown across. In particular, when the targeted bugs do not manifest *short-circuiting* (where bug encounter halts execution, e.g. a null-pointer exception), then buggy executions can be encoded as normal ones and subsequently detected compositionally using CISL-Par. For instance, when a data-agnostic data race is encountered, execution is not halted (though program behaviour may be undefined), and thus data races can be encoded as normal executions and detected by CISL-Par. By contrast, in the case of data-agnostic errors such as null-pointer exceptions, the execution is halted (i.e. short-circuited) and thus can no longer be encoded as normal executions that terminate. As such, *CISL cannot detect data-agnostic bugs with short-circuiting semantics compositionally.*

$$dom(\mathcal{G}_1) = \{\alpha_1, \alpha_2\} \qquad dom(\mathcal{G}_2) = \{\alpha_1', \alpha_2'\} \qquad \mathcal{R}_1 \triangleq \mathcal{G}_2 \qquad \mathcal{R}_2 \triangleq \mathcal{G}_1 \qquad \theta \triangleq [\alpha_1, \alpha_2, \alpha_1', \alpha_2']$$

$$\mathcal{G}_1(\alpha_1) \triangleq (x \Mapsto l_x * l_x \mapsto v_x,\ ok,\ x \Mapsto l_x * l_x \not\mapsto) \qquad \mathcal{G}_2(\alpha_1') \triangleq (z \Mapsto l_z * l_z \mapsto 1,\ ok,\ z \Mapsto l_z * l_z \mapsto 1)$$

$$\mathcal{G}_1(\alpha_2) \triangleq (z \Mapsto l_z * l_z \mapsto v_z,\ ok,\ z \Mapsto l_z * l_z \mapsto 1) \qquad \mathcal{G}_2(\alpha_2') \triangleq (x \Mapsto l_x * l_x \not\mapsto,\ er,\ x \Mapsto l_x * l_x \not\mapsto)$$

$$\emptyset, \mathcal{G}_1 \cup \mathcal{G}_2, \{[\,]\} \vdash \left[a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}\right] \quad // \textsc{Par}$$

$$\mathcal{R}_1, \mathcal{G}_1,$$
$$\{[\,]\} \vdash \left[\boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}\right]$$
$$1.\ \mathsf{free}(x); \quad // \textsc{Atom}, \text{MS-}\textsc{Free}$$
$$\{[\alpha_1]\} \vdash \left[ok\colon \boxed{\begin{array}{l} x \Mapsto l_x * l_x \not\mapsto \\ * z \Mapsto l_z * l_z \mapsto v_z \end{array}}\right]$$
$$2.\ [z] := 1; \quad // \textsc{Atom}, \text{MS-}\textsc{Write}$$
$$\{[\alpha_1, \alpha_2]\} \vdash \left[ok\colon \boxed{\begin{array}{l} x \Mapsto l_x * l_x \not\mapsto \\ * z \Mapsto l_z * l_z \mapsto 1 \end{array}}\right]$$
$$3.\ // \textsc{EnvR}$$
$$\{[\alpha_1, \alpha_2, \alpha_1']\} \vdash \left[ok\colon \boxed{\begin{array}{l} x \Mapsto l_x * l_x \not\mapsto \\ * z \Mapsto l_z * l_z \mapsto 1 \end{array}}\right]$$
$$4.\ // \textsc{EnvR}$$
$$\{\theta\} \vdash \left[er\colon \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$$

$$\|$$

$$\mathcal{R}_2, \mathcal{G}_2,$$
$$\{[\,]\} \vdash \left[a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}\right]$$
$$5.\ // \textsc{EnvL}$$
$$\{[\alpha_1]\} \vdash \left[ok\colon a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto v_z}\right]$$
$$6.\ // \textsc{EnvL}$$
$$\{[\alpha_1, \alpha_2]\} \vdash \left[ok\colon a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$$
$$7.\ a := 0; \quad // \textsc{AtomLocal}, \text{MS-}\textsc{AssignVal}$$
$$\{[\alpha_1, \alpha_2]\} \vdash \left[ok\colon a \Mapsto 0 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$$
$$8.\ a := [z]; \quad // \textsc{Atom}, \text{MS-}\textsc{Read}$$
$$\{[\alpha_1, \alpha_2, \alpha_1']\} \vdash \left[ok\colon a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$$
$$9.\ \mathsf{if}\ (a = 1)\ [x] := 1 \quad // \textsc{Atom}, \text{MS-}\textsc{WriteUAF}$$
$$\{\theta\} \vdash \left[er\colon a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$$

$$\emptyset, \mathcal{G}_1 \cup \mathcal{G}_2, \{\theta\} \vdash \left[er\colon a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}\right]$$

Figure 1 CASL proof of DataDep; the // denote CASL rules applied at each step. The $\mathcal{R}_1, \mathcal{G}_1$ and $\mathcal{R}_2, \mathcal{G}_2$ are not repeated at each step as they are unchanged.

More significantly, however, CISL is altogether *unable to detect data-dependent bugs compositionally*. Consider the data-dependent use-after-free bug at L in DataDep. As discussed, this bug occurs when τ_2 is executed after τ_1 is fully executed (i.e. 1 is written to z and x is deallocated). That is, for τ_2 to read 1 for z it must somehow infer that τ_1 writes 1 to z; this is not possible without having knowledge of the environment. This is reminiscent of *rely-guarantee* (RG) reasoning [11], where the environment behaviour is abstracted as a relation describing how it may manipulate the state. As RG only supports global and not compositional reasoning about states, RGSep [20] was developed by combining RG with separation logic to support state compositionality. We thus develop CASL as an under-approximate analogue of RGSep for bug catching (see p. 7 for a discussion on RGSep/RG).

2.1 CASL for Compositional Bug Detection

In CASL we prove under-approximate triples of the form $\mathcal{R}, \mathcal{G}, \Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]$, stating that every post-*world* $w_q \in Q$ is reached by running C on some pre-world $w_p \in P$, with $\mathcal{R}$, $\mathcal{G}$ and Θ described shortly. Each CASL world w is a pair (l, g), where $l \in \textsc{State}$ is the *local* state not accessible by the environment, while $g \in \textsc{State}$ is the *shared* (global) state accessible by all threads. We define CASL in a general, parametric way that can be instantiated for different use cases. As such, the choice of the underlying states, State, is a parameter to be instantiated accordingly. For instance, in what follows we instantiate CASL to detect the use-after-free bug in DataDep, where we define states as $\textsc{State} \triangleq \textsc{Stack} \times \textsc{Heap}$ (see §3), i.e. each state comprises a variable store and a heap.

For better readability, we use P, Q, R as meta-variable for sets of worlds and p, q, r for sets of states. We write $p * \boxed{q}$ for sets of worlds (l, g) where the local state is given by p $(l \in p)$ and the shared state is given by q $(g \in q)$. Given P and Q describing e.g. the worlds of two different threads, the composition $P * Q$ is defined component-wise on the local and shared states. More concretely, as local states are thread-private, they are combined via the composition operator $*$ on states in State (also supplied as a CASL parameter). On the other hand, as shared states are globally visible to all threads, the views of different threads of the shared state must agree and thus shared states are combined via conjunction $(\wedge)$. That is, given $P \triangleq p * \boxed{p'}$ and $Q \triangleq q * \boxed{q'}$, then $P * Q \triangleq p * q * \boxed{p' \wedge q'}$.

The *rely* relation, $\mathcal{R}$, describes how the environment threads may access/update the shared state, while the *guarantee* relation, $\mathcal{G}$, describes how the threads in C may do so. Specifically, both $\mathcal{R}$ and $\mathcal{G}$ are maps of *actions*: given $\mathcal{G}(\alpha) \triangleq (p, \epsilon, q)$, the α denotes an *action identifier* and (p, ϵ, q) denotes its effect, where p, q are sets of shared states and ϵ is an exit condition. Lastly, Θ denotes a set of *traces* (interleavings), such that each trace $\theta \in \Theta$ is a sequence of actions taken by the threads in C or the environment, i.e. the actions in $dom(\mathcal{G})$ and $dom(\mathcal{R})$. In particular, $\mathcal{R}, \mathcal{G}, \Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]$ states that for all traces $\theta \in \Theta$, each world in Q is reachable by executing C on some world in P culminating in θ, where the effects of the threads in C (resp. in the environment of C) on the shared state are given by $\mathcal{G}$ and $\mathcal{R}$, respectively. We shortly elaborate on this through an example.

CASL for Detecting Data-Dependent Bugs. Although CASL can detect all bugs identified by Raad et al. [18], we focus on using CASL for data-dependent bugs as they cannot be handled by the state-of-the-art CISL framework. In Fig. 1 we present a CASL proof sketch of the bug in DataDep. Let us write τ_1 and τ_2 for the left and right threads in Fig. 1, respectively. Variables x and z are accessed by both threads and are thus *shared*, whereas a is accessed by τ_2 only and is *local*. Similarly, heap locations l_x and l_z (recorded in x and z) are shared as they are accessed by both threads. This is denoted by $P_2 \triangleq a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}$ in the pre-condition of τ_2 in Fig. 1, describing worlds in which the local state is $a \Mapsto v_a$ (stating that stack variable a records value v_a), and the global state is $x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z$ – note that we use the $\Mapsto$ and $\mapsto$ arrows for stack and heap resources, respectively. By contrast, the τ_1 precondition is $P_1 \triangleq \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}$, comprising only shared resources and no local resources.

The actions in $\mathcal{G}_1$ (resp. $\mathcal{G}_2$), defined at the top of Fig. 1, describe the effect of τ_1 (resp. τ_2) on the shared state. For instance, $\mathcal{G}_1(\alpha_1)$ describes executing $\mathsf{free}(x)$ by τ_1: when the shared state contains $x \Mapsto l_x * l_x \mapsto v_x$, i.e. a *sub-part* of the shared state satisfies $x \Mapsto l_x * l_x \mapsto v_x$, then $\mathsf{free}(x)$ terminates normally (ok) and deallocates x, updating this sub-part to $x \Mapsto l_x * l_x \not\mapsto$, denoting that l_x is deallocated. Dually, the actions in $\mathcal{R}_1$ (resp. $\mathcal{R}_2$) describe the effect of the threads in the environment of τ_1 (resp. τ_2); e.g. as the environment of τ_1 comprises τ_2 only and $\mathcal{G}_2$ describes the effect of τ_2 on the shared state, we have $\mathcal{R}_1 \triangleq \mathcal{G}_2$.

Let us first consider analysing τ_2 in isolation, ignoring the // annotations for now (these become clear once we present the CASL proof rules in §3). Recall that in order to detect the use-after-free bug at L, thread τ_2 must account for an interleaving in which τ_1 executes both its instructions before τ_2 proceeds with its execution. That is, τ_2 may *assume* that τ_1 executes the actions associated with α_1 and α_2, as defined in $\mathcal{R}_2$. Note that after each environment action (in $\mathcal{R}_2$) we extend the trace to record the associated action (we elaborate on why this is needed below): starting from the empty trace $[]$, we subsequently update it to $[\alpha_1]$ and $[\alpha_1, \alpha_2]$ to record the environment actions assumed to have executed. Thread τ_2 then executes the (local) assignment instruction $a := 0$ (line 7) which accesses its local state $(a \Mapsto v_a)$ only. Subsequently, it proceeds to execute its instructions by accessing/updating the shared state as prescribed in $\mathcal{G}_2$: it 1) takes action α'_1 associated with executing $a := [z]$, whereby it reads from the heap location pointed to by z (i.e. l_z) and stores it in a; and then 2) takes action α_2' associated with executing $[x] := 1$, where it attempts to write to location l_x pointed to by x and arrives at a use-after-free error as l_x is deallocated, yielding $Q_2 \triangleq a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}$. Note that after each $\mathcal{G}_2$ action α the trace is extended with α, culminating in trace θ (defined at the top of Fig. 1). That is, each time a thread accesses the *shared* state it must do so through an action in its guarantee and record it in its trace. By contrast, when the instruction effect is limited to its *local* state (e.g. line 7 of τ_2), it may be executed freely, without consulting the guarantee or recording an action.

We next analyse τ_1 in isolation: τ_1 executes its two instructions as given by α_1 and α_2 in $\mathcal{G}_1$, updating the trace to $[\alpha_1, \alpha_2]$. It then assumes that τ_2 in its environment executes its actions (in $\mathcal{R}_1$), resulting in θ and yielding $Q_1 \triangleq \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}$. Note that τ_1 may assume that the environment action α_2' executes *erroneously*, as described in $\mathcal{R}_1(\alpha_2')$.

Finally, we reason about the full program using the CASL *parallel composition* rule, Par (in Fig. 3), stating that if we prove $\mathcal{R}_1, \mathcal{G}_1, \Theta_1 \vdash [P_1]$ C_1 $[\epsilon : Q_1]$ and separately $\mathcal{R}_2, \mathcal{G}_2, \Theta_2 \vdash [P_2]$ C_2 $[\epsilon : Q_2]$, then we can prove $\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2, \Theta_1 \cap \Theta_2 \vdash [P_1 * P_2]$ $\mathsf{C}_1 \,||\, \mathsf{C}_2$ $[\epsilon : Q_1 * Q_2]$ for the concurrent program $\mathsf{C}_1 \,||\, \mathsf{C}_2$. In other words, (1) the pre-condition (resp. post-conditions) of $\mathsf{C}_1 \,||\, \mathsf{C}_2$ is given by composing the pre-conditions (resp. post-conditions) of its constituent threads, namely $P_1 * P_2$ (resp. $Q_1 * Q_2$); (2) the effect of $\mathsf{C}_1 \,||\, \mathsf{C}_2$ on the shared state is the union of their respective effect (i.e. $\mathcal{G}_1 \cup \mathcal{G}_2$); (3) the effect of the $\mathsf{C}_1 \,||\, \mathsf{C}_2$ environment on the shared state is the effect of the threads in the environment of both C_1 and C_2 (i.e. $\mathcal{R}_1 \cap \mathcal{R}_2$); and (4) the traces generated by $\mathsf{C}_1 \,||\, \mathsf{C}_2$ are those generated by both C_1 and C_2 (i.e. $\Theta_1 \cap \Theta_2$).

Returning to Fig. 1, we use Par to reason about the full program. Let C_1 and C_2 denote the programs in the left and right threads, respectively. (1) Starting from $P \triangleq a \Mapsto v_a * \boxed{x \Mapsto l_x * l_x \mapsto v_x * z \Mapsto l_z * l_z \mapsto v_z}$, we split P as $P_1 * P_2$ (i.e. $P = P_1 * P_2$) and pass P_1 (resp. P_2) to τ_1 (resp. τ_2). (2) We analyse C_1 and C_2 in isolation and derive $\mathcal{R}_1, \mathcal{G}_1, \{\theta\} \vdash [P_1]$ C_1 $[er : Q_1]$ and $\mathcal{R}_2, \mathcal{G}_2, \{\theta\} \vdash [P_2]$ C_2 $[er : Q_2]$. (3) We use Par to combine the two triples and derive $\emptyset, \mathcal{G}_1 \cup \mathcal{G}_2, \{\theta\} \vdash [P]$ $\mathsf{C}_1 \,||\, \mathsf{C}_2$ $[er : Q]$ with $Q \triangleq a \Mapsto 1 * \boxed{x \Mapsto l_x * l_x \not\mapsto * z \Mapsto l_z * l_z \mapsto 1}$.

CISL versus CASL. In contrast to CISL-Par where we can only derive normal (ok) triples (and thus inevitably must encode erroneous behaviours as normal ones if possible), the CASL Par rule makes no such stipulation ($\epsilon = ok$ or $\epsilon \in \textsc{ErExit}$) and allows deriving both normal *and* erroneous triples. More significantly, a CISL triple $[P]$ C $[\epsilon : Q]$ executed by a thread τ only allows τ to take actions (updating the state) by executing C, i.e. only allows actions executed by τ itself and not those of other threads in the environment (executing another program C'). This is also the case for all *correctness* triples in over-approximate settings, e.g. RGSep and RG. By contrast, CASL triples additionally allow τ to *take a particular action by an environment thread*, as specified by rely, thereby allowing one to consider a specific interleaving (see the EnvL, EnvR and EnvEr rules in Fig. 3). This ability to *assume a specific execution by the environment* is missing from CISL. This is a crucial insight for data-dependent bugs that depend on certain data exchange/synchronisation between threads.

Recording Traces. Note that when taking a thread action (e.g. at line 1 in Fig. 1), the executing thread τ must adhere to the behaviour in its guarantee *and* additionally witness the action taken by executing corresponding instructions; this is captured by the CASL Atom rule. That is, the guarantee denotes what τ *can* do, and provides no assurance that τ does carry out those actions. This assurance is witnessed by executing corresponding instructions, e.g. τ_1 in Fig. 1 must execute $\mathsf{free}(x)$ on line 1 when taking α_1. By contrast, when τ takes an environment action (e.g. at line 3 in Fig. 1), it simply assumes the environment will take this action without witnessing it. That is, when reasoning about τ in isolation we *assume a particular interleaving* and show a given world is reachable under that interleaving. Therefore, the correctness of the compositional reasoning is contingent on the environment fulfilling this assumption by adhering to the *same interleaving.* This is indeed why we record θ, i.e. to ensure all threads assume the same sequence of actions on the shared state. As mentioned above, $\mathcal{R}, \mathcal{G}$ specify how the *shared* state is manipulated, and have no bearing on *thread-local* states. As such, we record no trace actions for instructions that only manipulate the local state (e.g. line 7 in Fig. 1); this is captured by the CASL AtomLocal rule.

Note that the Θ component of CASL is absent in its over-approximate counterpart RGSep. This is because in the *correctness* setting of RGSep one must prove a program is correct for *all interleavings* and it is not needed to record the interleavings considered. By contrast, in the *incorrectness* setting of CASL our aim is to show the occurrence of a bug under *certain interleavings* and thus we record them to ensure their feasibility: if a thread assumes a given interleaving θ, we must ensure that θ is a feasible interleaving for all concurrent threads.

RGSep versus RG. We develop CASL as an under-approximate analogue of RGSep [20] rather than RG [11]. We initially developed CASL as an under-approximate analogue of RG; however, the lack of support for local reasoning led to rather verbose proofs. Specifically, as discussed above and as we show in §4, the CASL AtomLocal rule allows local reasoning on thread-local resources without accounting for them in the recorded traces. By contrast, in RG there is no thread-local state and the entire state is shared (accessible by all threads). Hence, were we to base CASL on RG, we could only support the Atom rule and not the local AtomLocal variant, and thus every single action by each thread would have to be recorded in the trace. This not only leads to verbose proofs (with long traces), but it is also somewhat counter-intuitive. Specifically, thread-local computations (e.g. on thread-local registers) have no bearing on the behaviour of other threads and need not be reflected in the global trace. We present our original RG-based development [19, §E and §F] for the interested reader.

2.2 CASL for Compositional Exploit Detection

In practice, software attacks attempt to escalate privileges (e.g. Log4j) or steal credentials (e.g. Heartbleed [8]) using an *adversarial* program written by a security expert. That is, attackers typically use an adversarial program to interact with a codebase and exploit its vulnerabilities. Therefore, we can model a vulnerable program $\mathsf{C_v}$ and its adversary (attacker) $\mathsf{C_a}$ as the *concurrent* program $\mathsf{C_a} \,||\, \mathsf{C_v}$, and use CASL to detect vulnerabilities in $\mathsf{C_v}$. Vulnerabilities often fall into the *data-dependent* category, where the vulnerable program $\mathsf{C_v}$ receives an input from the adversary $\mathsf{C_a}$, and that input determines the next steps in the execution of $\mathsf{C_v}$, i.e. $\mathsf{C_a}$ affects the control flow of $\mathsf{C_v}$. Hence, existing under-approximate techniques such as CISL cannot detect such exploits, while the compositional techniques of CASL for detecting data-dependent bugs is ideally-suited for them. Indeed, to our knowledge CASL is the *first* formal, under-approximate theory that enables exploit detection. Thanks to the compositional nature of CASL, the approaches described here can be used to build *scalable* tools for exploit detection, as we discuss below. Moreover, by virtue of its under-approximate nature and built-in *no-false-positives* theorem, exploits detected by CASL are *certified* in that they are guaranteed to reveal true vulnerabilities.

In what follows we present an example of an information disclosure attack. Later we show how we use CASL to detect several classes of exploits, including: 1) *information disclosure attacks* on stacks (§4) and 2) heaps in the technical appendix [19, §C] to uncover sensitive data, e.g. Heartbleed [8]; and 3) *memory safety attacks* [19, §D], e.g. zero allocation [21].

Hereafter, we write $\mathsf{C_a}$ and $\mathsf{C_v}$ for the adversarial and vulnerable programs, respectively; and write τ_a and τ_v for the threads running $\mathsf{C_a}$ and $\mathsf{C_v}$, respectively. We represent exploits as $\mathsf{C_a} \,||\, \mathsf{C_v}$, positioning $\mathsf{C_a}$ and $\mathsf{C_v}$ as the left and right threads, respectively. As we discuss below, we model communication between τ_a and τ_v over a *shared channel* c, where each party can transmit (send/receive) information over c using the send and recv instructions.

```
                || local $sec := *$;
                || local $w[8] := \{0\}$;
send$(c, 8)$;   || recv$(c, x)$;
recv$(c, y)$;   || if $(x \leq 8)$              (InfDis)
                ||     $z := w[x]$;
                ||     send$(c, z)$;
```

Information Disclosure Attacks. Consider the InfDis example on the right, where τ_v (the vulnerable thread) allocates two variables on the stack: *sec*, denoting a secret initialised with a non-deterministic value $(*)$, and array w of size 8 initialised to 0. As per stack allocation, *sec* and w are allocated *contiguously* from the top of the stack. That is, when the top of the stack is denoted by top, then *sec* occupies the first unitof the stack (at top) and w occupies the next 8 units (between top-1 and top-8). In other words, w starts at top-8 and thus $w[i]$ resides at top$-8+i$.

The τ_v then receives x from τ_a, retrieves the x^{th} entry in w and sends it to τ_a over c. Specifically, τ_v first checks that x is valid (within bounds) via $x \leq 8$. However, as arrays are indexed from 0, for x to be valid we must have $x < 8$ instead, and thus this check is insufficient. That is, when τ_a sends 8 over c (send$(c, 8)$), then τ_v receives 8 on c and stores it in x (recv(c, x)), i.e. $x{=}8$, resulting in an out-of-bounds access ($z := w[x]$). As such, since $w[i]$ resides at top$-8+i$, $x{=}8$ and *sec* is at top, accessing $w[x]$ inadvertently retrieves the secret value *sec*, stores it in z, which is subsequently sent to τ_a over c, disclosing *sec* to τ_a!

CASL for Scalable Exploit Detection. In the over-approximate setting proving *correctness* (absence of bugs), a key challenge of developing *scalable* analysis tools lies in the need to consider *all* possible interleavings and establish bug freedom for all interleavings. In the under-approximate setting proving *incorrectness* (presence of bugs), this task is somewhat easier: it suffices to find *some* buggy interleaving. Nonetheless, in the absence of heuristics guiding the search for buggy interleavings, one must examine each interleaving to find buggy ones. Therefore, in the worst case one may have to consider all interleavings.

When using CASL to detect data-dependent bugs, the problem of identifying buggy interleavings amounts to determining *when* to account for environment actions. For instance, detecting the bug in Fig. 1 relied on accounting for the actions of the left thread at lines 5 and 6 prior to reading from z. Therefore, the scalability of a CASL-based bug detection tool hinges on developing heuristics that determine when to apply environment actions.

In the general case, where all threads may access any and all shared data (e.g. in DataDep), developing such heuristics may require sophisticated analysis of the synchronisation patterns used. However, in the case of exploits (e.g. in InfDis), the adversary and the vulnerable programs operate on mostly separate states, with the shared state comprising a shared channel (c) only, accessed through send and recv. In other words, the program *syntax* (send and recv instructions) provides a simple heuristic prescribing when the environment takes an action. Specifically, the computation carried out by τ_v is mostly *local* and does not affect the shared state c (i.e. by instructions other than send/recv); as discussed, such local steps need not be reflected in the trace and τ_a need not account for them. Moreover, when τ_v encounters a recv$(c, -)$ instruction, it must first assume the environment (τ_a) takes an action and sends a message over c to be subsequently received by τ_{v}. This leads to a *simple heuristic*: take an environment action prior to executing recv. We believe this observation can pave the way towards scalable exploit detection, underpinned by CASL and benefiting from its no-false-positives guarantee, certifying that the exploits detected are true positives.

3 CASL: A General Framework for Bug Detection

We present the general theory of the CASL framework for detecting concurrency bugs. We develop CASL in a *parametric* fashion, in that CASL may be instantiated for detecting bugs and exploits in a multitude of contexts. CASL is instantiated by supplying it with the specified parameters; the soundness of the instantiated CASL reasoning is then guaranteed *for free* from the soundness of the framework (see Theorem 2). We present the CASL ingredients as well as the parameters it is to be supplied with upon instantiation.

CASL Programming Language. The CASL language is parametrised by a set of *atoms*, $\textsc{Atom}$, ranged over by $\mathbf{a}$. For instance, our CASL instance for detecting memory safety bugs [19, §D] includes atoms for accessing the heap. This allows us to instantiate CASL for different scenarios without changing its underlying meta-theory. Our language is given by the C grammar below, and includes atoms $(\mathbf{a})$, skip, sequential composition $(\mathsf{C}_1; \mathsf{C}_2)$, non-deterministic choice $(\mathsf{C}_1 + \mathsf{C}_2)$, loops $(\mathsf{C}^\star)$ and parallel composition $(\mathsf{C}_1 \,||\, \mathsf{C}_2)$.

$$\textsc{Comm} \ni \mathsf{C} ::= \mathbf{a} \mid \mathsf{skip} \mid \mathsf{C}_1; \mathsf{C}_2 \mid \mathsf{C}_1 + \mathsf{C}_2 \mid \mathsf{C}^\star \mid \mathsf{C}_1 \,||\, \mathsf{C}_2$$

CASL States and Worlds. Reasoning frameworks [12, 18] typically reason at the level of high-level states, equipped with additional instrumentation to support diverse reasoning principles. In the frameworks based on separation logic, high-level states are modelled by a *partial commutative monoid* (PCM) of the form $(\textsc{State}, \circ, \textsc{State}_0)$, where $\textsc{State}$ denotes the set of *states*; $\circ : \textsc{State} \times \textsc{State} \rightharpoonup \textsc{State}$ denotes the partial, commutative and associative *state composition function*; and $\textsc{State}_0 \subseteq \textsc{State}$ denotes the set of unit states. Two states $l_1, l_2 \in \textsc{State}$ are *compatible*, written $l_1 \,\#\, l_2$, if their composition is defined: $l_1 \,\#\, l_2 \overset{\text{def}}{\iff} \exists l.\ l{=}l_1 \circ l_2$. Once CASL is instantiated with the desired state PCM, we define the notion of *worlds*, $\textsc{World}$, comprising pairs of states of the form (l, g), where $l \in \textsc{State}$ is the *local state* accessible only by the current thread(s), and $g \in \textsc{State}$ is the *shared* (global) state accessible by all threads (including those in the environment), provided that (l, g) is *well-formed.* A pair (l, g) is well-formed if the local and shared states are compatible $(l \,\#\, g)$.

▶ **Definition 1** (Worlds). *Assume a* PCM for states, $(\textsc{State}, \circ, \textsc{State}_0)$. *The set of* worlds *is* $\textsc{World} \triangleq \left\{(l, g) \in \textsc{State} \times \textsc{State} \,\middle|\, l \,\#\, g\right\}$. World composition, $\bullet : \textsc{World} \times \textsc{World} \rightharpoonup \textsc{World}$, *is defined component-wise,* $\bullet \triangleq (\circ, \circ_=)$, *where* $g \circ_= g' \triangleq g$ *when* $g{=}g'$, *and is otherwise undefined. The* world unit set *is* $\textsc{World}_0 \triangleq \left\{(l_0, g) \in \textsc{World} \,\middle|\, l_0 \in \textsc{State}_0 \wedge g \in \textsc{State}\right\}$.

Notation. We use p, q, r as metavariables for state sets (in $\mathcal{P}(\textsc{State})$), and P, Q, R as metavariables for world sets (in $\mathcal{P}(\textsc{World})$). We write $P * Q$ for $\left\{w \bullet w' \,\middle|\, w \in P \wedge w' \in Q\right\}$; $P \wedge Q$ for $P \cap Q$; $P \vee Q$ for $P \cup Q$; false for $\emptyset$; and true for $\mathcal{P}(\textsc{World})$. We write $p * \boxed{q}$ for $\left\{(l, g) \in \textsc{World} \,\middle|\, l \in p \wedge g \in q\right\}$. When clear from the context, we lift p, q, r to sets of worlds with arbitrary shared states; e.g. p denotes a set of worlds (l, g), where $l \in p$ and $g \in \textsc{State}$.

$$\alpha \in \textsc{AId} \qquad \mathcal{R}, \mathcal{G} \in \textsc{AMap} \triangleq \textsc{AId} \rightharpoonup \mathcal{P}(\textsc{State}) \times \textsc{Exit} \times \mathcal{P}(\textsc{State}) \qquad \Theta \in \mathcal{P}(\textsc{Trace})$$

$$\theta \in \textsc{Trace} \triangleq \textsc{List}\langle \textsc{AId} \rangle \qquad \Theta_0 \triangleq \{[\,]\} \qquad \Theta_1 +\!\!+ \Theta_2 \triangleq \left\{ \theta_1 +\!\!+ \theta_2 \,\middle|\, \theta_1 \in \Theta_1 \land \theta_2 \in \Theta_2 \right\}$$

$$\alpha :: \Theta \triangleq \left\{ \alpha :: \theta \,\middle|\, \theta \in \Theta \right\} \qquad \mathsf{dsj}(\mathcal{R}, \mathcal{G}) \overset{\text{def}}{\iff} dom(\mathcal{R}) \cap dom(\mathcal{G}) = \emptyset$$

$$\mathcal{R}_1 \subseteq \mathcal{R}_2 \overset{\text{def}}{\iff} dom(\mathcal{R}_1) \subseteq dom(\mathcal{R}_2) \land \forall \alpha \in dom(\mathcal{R}_1).\, \mathcal{R}_1(\alpha) = \mathcal{R}_2(\alpha)$$

$$\mathcal{R}' \preccurlyeq_\theta \mathcal{R} \overset{\text{def}}{\iff} \forall \alpha \in \theta \cap dom(\mathcal{R}').\ \mathcal{R}'(\alpha) = \mathcal{R}(\alpha) \qquad \mathcal{R}' \preccurlyeq_\Theta \mathcal{R} \overset{\text{def}}{\iff} \forall \theta \in \Theta.\ \mathcal{R}' \preccurlyeq_\theta \mathcal{R}$$

$$\mathsf{wf}(\mathcal{R}, \mathcal{G}) \overset{\text{def}}{\iff} \mathsf{dsj}(\mathcal{R}, \mathcal{G}) \land \forall \alpha \in dom(\mathcal{R}), p, q, l.\, \mathcal{R}(\alpha) = (p, -, q) \land q * \{l\} \neq \emptyset \Rightarrow p * \{l\} \neq \emptyset$$

Figure 2 The CASL model definitions.

Error Conditions and Atomic Axioms. CASL uses under-approximate triples [16, 17, 18] of the form $\mathcal{R}, \mathcal{G}, \Theta \vdash [p]\ \mathsf{C}\ [\epsilon : q]$, where $\epsilon \in \textsc{Exit} \triangleq \{ok\} \uplus \textsc{ErExit}$ denotes an *exit condition*, indicating normal (ok) or erroneous execution ($\epsilon \in \textsc{ErExit}$). Erroneous conditions in $\textsc{ErExit}$ are reasoning-specific and are supplied as a parameter, e.g. npe for a null pointer exception.

We shortly define the under-approximate proof system of CASL. As atoms are a CASL parameter, the CASL proof system is accordingly parametrised by their set of under-approximate *axioms*, $\textsc{Axiom} \subseteq \mathcal{P}(\textsc{State}) \times \textsc{Atom} \times \textsc{Exit} \times \mathcal{P}(\textsc{State})$, describing how they may update states. Concretely, an atomic axiom is a tuple $(p, \mathbf{a}, \epsilon, q)$, where $p, q \in \mathcal{P}(\textsc{State})$, $\mathbf{a} \in \textsc{Atom}$ and $\epsilon \in \textsc{Exit}$. As we describe shortly, atomic axioms are then lifted to CASL proof rules (see Atom and AtomLocal), describing how atomic commands may modify worlds.

CASL Triples. A CASL triple $\mathcal{R}, \mathcal{G}, \Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]$ states that every world in Q can be reached under ϵ for every *witness trace* $\theta \in \Theta$ by executing C on some world in P. Moreover, at each step the actions of the current thread (executing C) and its environment adhere to $\mathcal{G}$ and $\mathcal{R}$, respectively. The $\mathcal{R}, \mathcal{G}$ are defined as *action maps* in Fig. 2, mapping each action $\alpha \in \textsc{AId}$ to a triple describing its behaviour. Compared to original rely/guarantee relations [20, 11], in CASL we record two additional components: 1) the exit condition (ϵ) indicating a normal or erroneous step; and 2) the action id (α) to identify actions uniquely. The latter allows us to construct a witness interleaving $\theta \in \textsc{Trace}$ as a list of actions (see Fig. 2). As discussed in §2, to avoid false positives, if we detect a bug assuming the environment takes action α, we must indeed witness the environment taking α. That is, if we detect a bug assuming the environment takes α but the environment cannot do so, then the bug is a false positive. Recording traces ensures each thread fulfils its assumptions, as we describe shortly.

Intuitively, each α corresponds to executing an atom that updates a *sub-part* of the shared state. Specifically, $\mathcal{G}(\alpha) = (p, \epsilon, q)$ (resp. $\mathcal{R}(\alpha) = (p, \epsilon, q)$) denotes that the current thread (resp. an environment thread) may take α and update a shared sub-state in p to one in q under ϵ, and in doing so it extends each trace in Θ with α. Moreover, the current thread may take α with $\mathcal{G}(\alpha) = (p, \epsilon, q)$ only if it executes an atom $\mathbf{a}$ with behaviour (p, ϵ, q), i.e. $(p, \mathbf{a}, \epsilon, q) \in \textsc{Axiom}$, thereby *witnessing* α. By contrast, this is not required for an environment action. As we describe below, this is because each thread witnesses the $\mathcal{G}$ actions it takes, and thus when combining threads (using the CASL Par rule described below), so long as they agree on the interleavings (traces) taken, then the actions recorded have been witnessed.

Lastly, we require $\mathcal{R}$, $\mathcal{G}$ to be *well-formed* ($\mathsf{wf}(\mathcal{R},\mathcal{G})$ in Fig. 2), stipulating that: 1) $\mathcal{R}$ and $\mathcal{G}$ be *disjoint*, $\mathsf{dsj}(\mathcal{R},\mathcal{G})$; and 2) the actions in $\mathcal{R}$ be *frame-preserving*: for all α with $\mathcal{R}(\alpha) = (p, -, q)$ and all states l, if l is compatible with q (i.e. $q * \{l\} \neq \emptyset$), then l is also compatible with p (i.e. $p * \{l\} \neq \emptyset$). Condition (1) allows us to attribute actions uniquely to threads (i.e. distinguish between $\mathcal{R}$ and $\mathcal{G}$ actions). Condition (2) is necessary for the CASL FRAME rule (see below), ensuring that applying an environment action does not inadvertently update the state in such a way that invalidates the resources in the frame. Note that we require no such condition on $\mathcal{G}$ actions. This is because as discussed, each $\mathcal{G}$ action taken is witnessed by executing an atom axiomatised in AXIOM; axioms in AXIOM must in turn be frame-preserving to ensure the soundness of CASL. That is, a $\mathcal{G}$ action is taken only if it is witnessed by an atom which is frame-preserving by definition (see SOUNDATOMS in [19, §A]).

CASL Proof Rules. We present the CASL proof rules in Fig. 3, where we assume the rely/guarantee relations in triple contexts are well-formed. SKIP states that executing skip leaves the worlds (P) unchanged and takes no actions, yielding a single empty trace $\Theta_0 \triangleq \{[\,]\}$. SEQ, SEQER, CHOICE, LOOP1, LOOP2 and BACKWARDSVARIANT are analogous to those of IL [16] with $S : \mathbb{N} \to \mathcal{P}(\text{WORLD})$. Note that in SEQ, the set of traces resulting from executing $\mathsf{C}_1; \mathsf{C}_2$ is given by $\Theta_1 {+\!\!+} \Theta_2$ (defined in Fig. 2) by point-wise combining the traces of C_1 and C_2.

ATOM describes how executing an atom $\mathbf{a}$ affects the shared state: when the local state is in p' and the shared state is in $p * f$, i.e. a sub-part of the shared state is in p, then executing $\mathbf{a}$ with $(p' * p, \mathbf{a}, \epsilon, q' * q) \in \text{AXIOM}$ updates the local state from p' to q' and the shared sub-part from p to q, provided that the effect on the shared state is given by a guarantee action α ($\mathcal{G}(\alpha) = (p, \epsilon, q)$). That is, the $\mathcal{G}$ action only captures the shared state, and the thread may update its local state freely. In doing so, we *witness* α and record it in the set of traces ($\{[\alpha]\}$). By contrast, ATOMLOCAL states that so long as executing $\mathbf{a}$ does not touch the shared state, it may update the local state arbitrarily, without recording an action.

ENVL, ENVR and ENVER are the ATOM counterparts in that they describe how the *environment* may update the shared state. Specifically, ENVL and ENVR state that the current thread may be interleaved by the environment. Given $\alpha \in \mathit{dom}(\mathcal{R})$, the current thread may execute C either *after* or *before* the environment takes action α, as captured by ENVL and ENVR, respectively. In the case of ENVL we further require that α (in $\mathit{dom}(\mathcal{R})$) denote a normal ($\mathit{ok}$) execution step, as otherwise the execution would short-circuit and the current thread could not execute C. Note that unlike in ATOM, the environment action α in ENVL and ENVR only updates the shared state; e.g. in ENVL the p sub-part of the shared state is updated to r and the local state p' is left unchanged. Analogously, ENVER states that executing C may terminate erroneously under er if it is interleaved by an *erroneous* step of the environment under er. That is, if the environment takes an erroneous step, the execution of the current thread is terminated, as per the short-circuiting semantics of errors.

Note that ATOM ensures action α is taken by the current thread (in $\mathcal{G}$) only when the thread witnesses it by executing a matching atom. By contrast, in ENVL, ENVR and ENVER we merely *assume* the environment takes action α in $\mathcal{R}$. As such, each thread locally ensures that it takes the guarantee actions in its traces.As shown in PAR, when joining the threads via parallel composition $\mathsf{C}_1 \,||\, \mathsf{C}_2$, we ensure their sets of traces agree: $\Theta_1 \cap \Theta_2 \neq \emptyset$. Moreover, to ensure we can attribute each action in traces to a unique thread, we require that $\mathcal{G}_1$ and $\mathcal{G}_2$ be disjoint ($\mathsf{dsj}(\mathcal{G}_1, \mathcal{G}_2)$, see Fig. 2). Finally, when τ_1 and τ_2 respectively denote the threads running C_1 and C_2, the $\mathcal{R}_1 \subseteq \mathcal{G}_2 \cup \mathcal{R}_2$ premise ensures when τ_1 attributes an action α to $\mathcal{R}_1$ (i.e. α is in $\mathcal{R}_1$), then α is an action of either τ_2 (i.e. α is in $\mathcal{G}_2$) or its environment (i.e. of a thread running concurrently with both τ_1 and τ_2); similarly for $\mathcal{R}_2 \subseteq \mathcal{G}_1 \cup \mathcal{R}_1$.

Skip

$$\mathcal{R},\mathcal{G},\Theta_0 \vdash [P]\ \mathsf{skip}\ [ok\colon P]$$

Seq

$$\frac{\mathcal{R},\mathcal{G},\Theta_1 \vdash [P]\ \mathsf{C}_1\ [ok\colon R] \quad \mathcal{R},\mathcal{G},\Theta_2 \vdash [R]\ \mathsf{C}_2\ [\epsilon : Q]}{\mathcal{R},\mathcal{G},\Theta_1 +\!\!+ \Theta_2 \vdash [P]\ \mathsf{C}_1;\mathsf{C}_2\ [\epsilon : Q]}$$

SeqEr

$$\frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1\ [er : Q] \quad er \in \textsc{ErExit}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1;\mathsf{C}_2\ [er\colon Q]}$$

Atom

$$\frac{\mathcal{G}(\alpha)=(p,\epsilon,q) \quad (p' * p, \mathbf{a}, \epsilon, q' * q) \in \textsc{Axiom}}{\mathcal{R},\mathcal{G},\{[\alpha]\} \vdash \left[p' * \boxed{p * f}\right]\ \mathbf{a}\ \left[\epsilon : q' * \boxed{q * f}\right]}$$

Loop1

$$\mathcal{R},\mathcal{G},\Theta_0 \vdash [P]\ \mathsf{C}^\star\ [ok\colon P]$$

Loop2

$$\frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}^\star;\mathsf{C}\ [\epsilon : Q]}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}^\star\ [\epsilon : Q]}$$

AtomLocal

$$\frac{(p, \mathbf{a}, ok, q) \in \textsc{Axiom}}{\mathcal{R},\mathcal{G},\{[\,]\} \vdash [p]\ \mathbf{a}\ [ok\colon q]}$$

BackwardsVariant

$$\frac{\forall k.\ \mathcal{R},\mathcal{G},\Theta \vdash [S(k)]\ \mathsf{C}\ [ok\colon S(k{+}1)] \qquad \forall n{>}0.\ \Theta_n = \Theta +\!\!+ \Theta_{n-1}}{\mathcal{R},\mathcal{G},\Theta_n \vdash [S(0)]\ \mathsf{C}\ [ok\colon S(n)]}$$

Choice

$$\frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_i\ [\epsilon : Q] \quad \text{for some } i \in \{1,2\}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1 + \mathsf{C}_2\ [\epsilon : Q]}$$

Comb

$$\frac{\mathcal{R},\mathcal{G},\Theta_1 \vdash [P]\ \mathsf{C}\ [\epsilon : Q] \quad \mathcal{R},\mathcal{G},\Theta_2 \vdash [P]\ \mathsf{C}\ [\epsilon : Q]}{\mathcal{R},\mathcal{G},\Theta_1 \cup \Theta_2 \vdash [P]\ \mathsf{C}\ [\epsilon : Q]}$$

EnvL

$$\frac{\mathcal{R}(\alpha)=(p,ok,r) \quad \mathcal{R},\mathcal{G},\Theta \vdash \left[p' * \boxed{r * f}\right]\ \mathsf{C}\ [\epsilon : Q]}{\mathcal{R},\mathcal{G},\alpha :: \Theta \vdash \left[p' * \boxed{p * f}\right]\ \mathsf{C}\ [\epsilon : Q]}$$

EnvR

$$\frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ \left[ok : r' * \boxed{r * f}\right] \quad \mathcal{R}(\alpha)=(r,\epsilon,q)}{\mathcal{R},\mathcal{G},\Theta +\!\!+ \{[\alpha]\} \vdash [P]\ \mathsf{C}\ \left[\epsilon : r' * \boxed{q * f}\right]}$$

EnvEr

$$\frac{\mathcal{R}(\alpha) = (p, er, q) \qquad er \in \textsc{ErExit}}{\mathcal{R},\mathcal{G},\{[\alpha]\} \vdash \left[\boxed{p * f}\right]\ \mathsf{C}\ \left[er\colon \boxed{q * f}\right]}$$

Frame

$$\frac{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q] \quad \mathsf{stable}(R, \mathcal{R} \cup \mathcal{G})}{\mathcal{R},\mathcal{G},\Theta \vdash [P * R]\ \mathsf{C}\ [\epsilon : Q * R]}$$

ParEr

$$\frac{\begin{array}{c}\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_i\ [er\colon Q] \ \text{for some } i \in \{1,2\} \\ er \in \textsc{ErExit} \qquad \Theta \sqsubseteq \mathcal{G}\end{array}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [er\colon Q]}$$

Cons

$$\frac{\begin{array}{c}P' \subseteq P \qquad \mathcal{R}',\mathcal{G}',\Theta' \vdash [P']\ \mathsf{C}\ [\epsilon : Q'] \qquad Q \subseteq Q' \\ \mathcal{R} \preccurlyeq_\Theta \mathcal{R}' \qquad \mathcal{G} \preccurlyeq_\Theta \mathcal{G}' \qquad \Theta \subseteq \Theta'\end{array}}{\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [\epsilon : Q]}$$

Par

$$\frac{\begin{array}{c}\mathcal{R}_1,\mathcal{G}_1,\Theta_1 \vdash [P_1]\ \mathsf{C}_1\ [\epsilon : Q_1] \qquad \mathcal{R}_2,\mathcal{G}_2,\Theta_2 \vdash [P_2]\ \mathsf{C}_2\ [\epsilon : Q_2] \\ \mathcal{R}_1 \subseteq \mathcal{G}_2 \cup \mathcal{R}_2 \qquad \mathcal{R}_2 \subseteq \mathcal{G}_1 \cup \mathcal{R}_1 \qquad \mathsf{dsj}(\mathcal{G}_1,\mathcal{G}_2) \qquad \Theta_1 \cap \Theta_2 \neq \emptyset\end{array}}{\mathcal{R}_1 \cap \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2, \Theta_1 \cap \Theta_2 \vdash [P_1 * P_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [\epsilon : Q_1 * Q_2]}$$

$$\begin{array}{rrl} \text{with} & \Theta \sqsubseteq \mathcal{G} & \overset{\text{def}}{\iff} \forall \theta \in \Theta.\ \theta \subseteq dom(\mathcal{G}) \\ \text{and} & \mathsf{stable}(R, \mathcal{R}) & \overset{\text{def}}{\iff} \forall (l,g) \in R, \alpha.\ \forall (p,-,q) \in \mathcal{R}(\alpha), g_q \in q, g_p \in p, g'.\ g = g_q \circ g' \Rightarrow (l, g_p \circ g') \in R \end{array}$$

Figure 3 The CASL proof rules, where $\mathcal{R}/\mathcal{G}$ relations in contexts are well-formed.

Observe that Par can be used for both normal and erroneous triples (i.e. for any ϵ) *compositionally*. This is in contrast to CISL, where only ok triples can be proved using CISL-Par, and thus bugs can be detected only if they can be encoded as ok (see §2). In other words, CISL cannot compositionally detect either data-agnostic bugs with short-circuiting semantics or data-dependent bugs altogether, while CASL can detect both data-agnostic and data-dependent bugs compositionally using Par, without the need to encode them as ok. This is because CASL captures the environment in $\mathcal{R}$, enabling compositional reasoning. In particular, even when we do not know the program in parallel, so long as its behaviour adheres to $\mathcal{R}$, we can detect an error: $\mathcal{R},\mathcal{G},\Theta \vdash [P]\ \mathsf{C}\ [er : Q]$ ensures the error is reachable as long as the environment adheres to $\mathcal{R}$, without knowing the program run in parallel to C.

ParEr is the concurrent analogue of SeqEr, describing the short-circuiting semantics of concurrent executions: given $i \in \{1, 2\}$, if running C_i in isolation results in an error, then running $\mathsf{C}_1 \,||\, \mathsf{C}_2$ also yields an error. The $\Theta \sqsubseteq \mathcal{G}$ premise (defined in Fig. 3) ensures the actions in Θ are from $\mathcal{G}$, i.e. taken by the current thread and not assumed to have been taken by the environment. Comb allows us to extend the traces: if the traces in both Θ_1 and Θ_2 witness the execution of C, then the traces in $\Theta_1 \cup \Theta_2$ also witness the execution of C.

Cons is the CASL rule of consequence. As with under-approximate logics [16, 17, 18], the post-worlds Q may shrink ($Q \subseteq Q'$) and the pre-worlds P may grow ($P' \subseteq P$). The traces may shrink ($\Theta \subseteq \Theta'$): if traces in Θ' witness executing C, then so do the traces in the smaller set Θ. Lastly, $\mathcal{R} \preccurlyeq_\Theta \mathcal{R}'$ (resp. $\mathcal{G} \preccurlyeq_\Theta \mathcal{G}'$) defined in Fig. 2 states that the rely (resp. guarantee) may *grow or shrink* so long as it preserves the behaviour of actions in Θ. This is in contrast to RG/RGSep where the rely may only shrink and the guarantee may only grow. This is because in RG/RGSep one must defensively prove correctness against *all* environment actions at *all program points*, i.e. for *all interleavings*. Therefore, if a program is correct under a larger environment (with more actions) $\mathcal{R}'$, then it is also correct under a smaller environment $\mathcal{R}$. In CASL, however, we show an outcome is reachable under a set of witness interleavings Θ. Hence, for traces in Θ to remain valid witnesses, the rely/guarantee may grow or shrink, so long as they faithfully reflect the behaviours of the actions in Θ.

Lastly, Frame states that if we show $\mathcal{R}, \mathcal{G}, \Theta \vdash [P]\, \mathsf{C}\, [\epsilon{:}Q]$, we can also show $\mathcal{R}, \mathcal{G}, \Theta \vdash [P * R]\, \mathsf{C}\, [\epsilon{:}Q * R]$, so long as the worlds in R are *stable* under $\mathcal{R}, \mathcal{G}$ ($\mathsf{stable}(R, \mathcal{R} \cup \mathcal{G})$, defined in Fig. 3), in that R accounts for possible updates. That is, given $(l, g) \in R$ and α with $(p, -, q) \in \mathcal{R}(\alpha) \cup \mathcal{G}(\alpha)$, if a sub-part g_q of the shared g is in q ($g = g_q \circ g'$ for some $g_q \in q$ and g'), then replacing g_q with an arbitrary $g_p \in p$ results in a world (i.e. $(l, g_p \circ g')$) also in R.

CASL Soundness. We define the formal interpretation of CASL triples via *semantic triples* of the form $\mathcal{R}, \mathcal{G}, \Theta \models [P]\ \mathsf{C}\ [\epsilon : Q]$ (see [19, §A]). We show CASL is sound by showing its triples in Fig. 3 induce valid semantics triples. We do this in the theorem below, with its proof in [19, §B].

▶ **Theorem 2** (Soundness). *For all $\mathcal{R}, \mathcal{G}, \Theta, p, \mathsf{C}, \epsilon, q$, if $\mathcal{R}, \mathcal{G}, \Theta \vdash [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 3, then $\mathcal{R}, \mathcal{G}, \Theta \models [p]\ \mathsf{C}\ [\epsilon : q]$ holds.*

4 CASL for Exploit Detection

We present $\text{CASL}_{\mathsf{ID}}$, a CASL instance for detecting *stack-based information disclosure* exploits. In the technical appendix [19] we present $\text{CASL}_{\mathsf{HID}}$ for detecting *heap-based information disclosure* exploits [19, §C] and $\text{CASL}_{\mathsf{MS}}$ for detecting *memory safety attacks* [19, §D].

The $\text{CASL}_{\mathsf{ID}}$ atomics, $\text{Atom}_{\mathsf{ID}}$, are below, where $\textsc{l} \in \mathbb{N}$ is a label, x, y are (local) variables, c is a shared channel and v is a value. They include assume statements and primitives for generating a random value $*$ ($\mathsf{local}\, x :=_\tau *$) used to model a secret value (e.g. a private key), declaring an array x of size n initialised with v ($\mathsf{local}\, x[n] :=_\tau \{v\}$), array assignment $\textsc{l}{:}\, x[k] :=_\tau y$, sending ($\mathsf{send}(c, x)$ and $\mathsf{send}(c, v)$) and receiving ($\mathsf{recv}(c, x)$) over channel c. As is standard, we encode $\mathsf{if}\ (b)\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2$ as $(\mathsf{assume}(b); \mathsf{C}_1) + (\mathsf{assume}(\neg b); \mathsf{C}_2)$.

$$\begin{aligned} \textsc{Atom}_{\mathsf{ID}} \ni \mathbf{a} ::= {} & \textsc{l}{:}\, \mathsf{assume}(b) \mid \textsc{l}{:}\, \mathsf{local}\, x :=_\tau * \mid \textsc{l}{:}\, \mathsf{local}\, x[k] :=_\tau \{v\} \mid \textsc{l}{:}\, x :=_\tau y[k] \\ & \mid \textsc{l}{:}\, \mathsf{send}(c, x)_\tau \mid \textsc{l}{:}\, \mathsf{send}(c, v)_\tau \mid \textsc{l}{:}\, \mathsf{recv}(c, x)_\tau \end{aligned}$$

$\text{CASL}_{\mathsf{ID}}$ States. A $\text{CASL}_{\mathsf{ID}}$ state, $(s, h, \mathbf{h})$, comprises a *variable stack* $s \in \textsc{Stack} \triangleq \textsc{Var} \rightharpoonup \widetilde{\textsc{Val}}$, mapping variables to *instrumented values*; a *heap* $h \in \textsc{Heap} \triangleq \textsc{Loc} \rightharpoonup (\widetilde{\textsc{Val}} \cup \textsc{List}\langle \widetilde{\textsc{Val}} \rangle)$, mapping shared locations (e.g. channel c) to (lists of) instrumented values; and a *ghost*

ID-VarSecret

$$[\mathbf{s}_\tau \vdash\!\dashrightarrow n]\ \textsc{l}\colon \mathsf{local}\, x :=_\tau * \ [ok\colon \mathbf{s}_\tau \vdash\!\dashrightarrow (n{+}1) * x = \mathsf{top}{-}n * x \Mapsto (v, \tau, 1)]$$

ID-VarArray

$$[\mathbf{s}_\tau \vdash\!\dashrightarrow n * k > 0]\ \textsc{l}\colon \mathsf{local}\, x[k] :=_\tau \{v\} \Big[ok\colon \mathbf{s}_\tau \vdash\!\dashrightarrow (n{+}k) * x = \mathsf{top}{-}(n{+}k{-}1) * \mathop{\Huge *}_{j=0}^{k-1} (x{+}j \Mapsto (v,\tau,0)) * k > 0\Big]$$

ID-ReadArray

$$[k \Mapsto (v, \tau_v, b) * y{+}v \Mapsto V_y * x \Mapsto -]\ \textsc{l}\colon x :=_\tau y[k]\ [ok\colon k \Mapsto (v, \tau_v, b) * y{+}v \Mapsto V_y * x \Mapsto V_y]$$

ID-SendVal

$$[c \mapsto L]\ \textsc{l}\colon \mathsf{send}(c,v)_\tau\ [ok\colon c \mapsto L \mathbin{+\!\!+} [(v,\tau,0)]]$$

ID-Send

$$[c \mapsto L * x \Mapsto V]\ \textsc{l}\colon \mathsf{send}(c,x)_\tau\ [ok\colon c \mapsto L \mathbin{+\!\!+} [V]]$$

ID-Recv

$$[c \mapsto [(v,\tau_t,\iota)] \mathbin{+\!\!+} L * x \Mapsto - * (\iota{=}0 \vee \tau \in \mathsf{Trust})]\ \textsc{l}\colon \mathsf{recv}(c, x)_\tau\ [ok\colon c \mapsto L * x \Mapsto (v,\tau_t,\iota) * (\iota{=}0 \vee \tau \in \mathsf{Trust})]$$

ID-RecvEr

$$[c \mapsto [(v, \tau_t, 1)] \mathbin{+\!\!+} L * \tau \notin \mathsf{Trust}]\ \textsc{l}\colon \mathsf{recv}(c, x)_\tau\ [er\colon c \mapsto [(v, \tau_t, 1)] \mathbin{+\!\!+} L * \tau \notin \mathsf{Trust}]$$

Figure 4 The CASL$_{\text{ID}}$ axioms.

heap $\mathbf{h} \in \text{GHeap} \triangleq (\{\mathbf{s}\} \times \text{TId}) \rightharpoonup \text{Val}$, tracking the stack size ($\mathbf{s}$). An instrumented value, $(v, \tau, \iota) \in \widetilde{\text{Val}} \triangleq \text{Val} \times \text{TId} \times \{0, 1\}$, comprises a value ($v$), its provenance ($\tau$, the thread from which v originated), and its *secret attribute* ($\iota \in \{0, 1\}$) denoting whether the value is secret (1) or not (0). We use x, y as metavariables for local variables, c for shared channels, v for values, L for value lists and V for instrumented values. State composition is defined as $(\uplus, \uplus, \uplus)$, where $\uplus$ denotes disjoint function union. The state unit set is $\{(\emptyset, \emptyset, \emptyset)\}$. We write $x \Mapsto V$ for states in which the stack comprises a single variable x mapped on to V and the heap and ghost heaps are empty, i.e. $\{([x \mapsto V], \emptyset, \emptyset)\}$. Similarly, we write $c \mapsto L$ for $\{(\emptyset, [c \mapsto L], \emptyset)\}$, and $\mathbf{s}_\tau \vdash\!\dashrightarrow v$ for $\{(\emptyset, \emptyset, [(\mathbf{s}, \tau) \mapsto v])\}$.

CASL$_{\text{ID}}$ Axioms. We present the CASL$_{\text{ID}}$ atomic axioms in Fig. 4. We assume that each variable declaration (via $\mathsf{local}\, x :=_\tau *$ and $\mathsf{local}\, x[n] :=_\tau \{v\}$) defines a *fresh* name, and thus avoid the need for variable renaming at declaration time. We assume the stack top is given by the constant top; thus when the stack of thread τ is of size n (i.e. $\mathbf{s}_\tau \vdash\!\dashrightarrow n$), the next empty stack spot is at $\mathsf{top}{-}n$. Executing $\textsc{l}\colon \mathsf{local}\, x :=_\tau *$ in ID-VarSecret increments the stack size ($\mathbf{s}_\tau \vdash\!\dashrightarrow n{+}1$), reserves the next empty spot for x and initialises x with a value (v) marked secret (1) with its provenance (thread τ). Analogously, ID-VarArray describes declaring an array of size k, where the next k spots are reserved for x (the $\mathop{\Large *}$ denotes $*$-iteration: $\mathop{\Large *}_{j=1}^{n} (x{+}j \Mapsto V) \triangleq x{+}1 \Mapsto V * \cdots * x{+}n \Mapsto V$). When k holds value v, ID-ReadArray reads the v^{th} entry of y (at $y{+}v$) in x. ID-SendVal extends the content of c with $(v, \tau, 0)$. ID-Recv describes *safe* data receipt (not leading to *information disclosure*), i.e. the value received is not secret ($\iota{=}0$) or the recipient is *trusted* ($\tau \in \mathsf{Trust} \triangleq \text{TId} \setminus \{\tau_{\mathsf{a}}\}$). By contrast, ID-RecvEr describes when receiving data leads to information disclosure, i.e. the value received is secret and the recipient is untrusted ($\tau \notin \mathsf{Trust}$), in which case the state is unchanged.

Example: InfDis. In Fig. 5 we present a CASL$_{\text{ID}}$ proof sketch of the information disclosure exploit in InfDis. The proof of the full program is given in Fig. 5a. Starting from $P_a * P_v$ with a singleton empty trace (Θ_0, defined in Fig. 2), we use Par to pass P_a and P_v respectively to τ_{a} and τ_{v}, analyse each thread in isolation, and combine their results (Q_a and Q_v) into $Q_a * Q_v$, with the two agreeing on the trace set Θ generated. Figures 5b and 5c show the proofs of τ_{a} and τ_{v}, respectively, where we have also defined their pre- and post-conditions.

$$\mathcal{R}_v(\alpha'_1) \triangleq (c\mapsto [\,], ok, c\mapsto [(n, \tau_{\mathsf{a}}, 0)]) \qquad \mathcal{R}_v(\alpha'_2) \triangleq (c\mapsto [(v, \tau, 1)], ok, c\mapsto [\,]) \qquad \mathcal{R}_a \triangleq \mathcal{G}_v \qquad \mathcal{G}_a \triangleq \mathcal{R}_v$$
$$\mathcal{G}_v(\alpha_1) \triangleq (c\mapsto [(n, \tau_{\mathsf{a}}, 0)], ok, c\mapsto [\,]) \qquad \mathcal{G}_v(\alpha_2) \triangleq (c\mapsto [\,], ok, c\mapsto (v, \tau, 1)) \qquad \Theta \triangleq \{[\alpha'_1, \alpha_1, \alpha_2, \alpha'_2]\}$$

$$\emptyset, \mathcal{G}_a \cup \mathcal{G}_v, \Theta_0 \vdash [P_a * P_v] \quad //\,\textsc{Par}$$

$$\mathcal{R}_a, \mathcal{G}_a, \Theta_0 \vdash [P_a]$$
$$\mathrm{L}'_1\colon \mathsf{send}(c, 8)_{\tau_{\mathsf{a}}}$$
$$\mathrm{L}'_2\colon \mathsf{recv}(c, y)_{\tau_{\mathsf{a}}}$$
$$\mathcal{R}_a, \mathcal{G}_a, \Theta \vdash [er\colon Q_a]$$

$$\Big\|$$

$$\mathcal{R}_v, \mathcal{G}_v, \Theta_0 \vdash [P_v]$$
$$\mathrm{L}_1\colon \mathsf{local}\ sec :=_{\tau_{\mathsf{v}}} *$$
$$\mathrm{L}_2\colon \mathsf{local}\ w[8] :=_{\tau_{\mathsf{v}}} \{v\}$$
$$\mathrm{L}_3\colon \mathsf{recv}(c, x)_{\tau_{\mathsf{v}}}$$
$$\mathrm{L}_4\colon z :=_{\tau_{\mathsf{v}}} w[x]$$
$$\mathrm{L}_5\colon \mathsf{send}(c, z)_{\tau_{\mathsf{v}}}$$
$$\mathcal{R}_v, \mathcal{G}_v, \Theta \vdash [er\colon Q_v]$$

$$\emptyset, \mathcal{G}_a \cup \mathcal{G}_v, \Theta \vdash [er\colon Q_a * Q_v]$$

(a)

$$\mathcal{R}_a, \mathcal{G}_a, \Theta_0 \vdash \left[P_a \triangleq \boxed{c\mapsto [\,]} * \tau_{\mathsf{a}} \notin \mathsf{Trust}\right]$$
$$\mathrm{L}'_1\colon \mathsf{send}(c, 8)_{\tau_{\mathsf{a}}} \quad //\,\textsc{Atom} + \textsc{ID-SendVal}$$
$$\mathcal{R}_a, \mathcal{G}_a, \{[\alpha'_1]\} \vdash \left[ok\colon \boxed{c\mapsto [(8, \tau_{\mathsf{a}}, 0)]} * \tau_{\mathsf{a}} \notin \mathsf{Trust}\right]$$
$$//\,\textsc{EnvL} \times 2$$
$$\mathcal{R}_a, \mathcal{G}_a, \{[\alpha'_1, \alpha_1, \alpha_2]\} \vdash \left[ok\colon \boxed{c\mapsto [(v, \tau_{\mathsf{v}}, 1)]} * \tau_{\mathsf{a}} \notin \mathsf{Trust}\right]$$
$$\mathrm{L}'_2\colon \mathsf{recv}(c, t)_{\tau_{\mathsf{a}}} \quad //\,\textsc{Atom} + \textsc{ID-RecvEr}$$
$$\mathcal{R}_a, \mathcal{G}_a, \Theta \vdash \left[er\colon Q_a \triangleq \boxed{c\mapsto [(v, \tau_{\mathsf{v}}, 1)]} * \tau_{\mathsf{a}} \notin \mathsf{Trust}\right]$$

(b)

$$\mathcal{R}_v, \mathcal{G}_v,$$
$$\Theta_0 \vdash \left[P \triangleq \mathbf{s}_{\tau_{\mathsf{v}}} \vdash\!\!\dashrightarrow 0 * x \Mapsto - * z \Mapsto - * \boxed{c\mapsto [\,]}\right]$$
$$\mathrm{L}_1\colon \mathsf{local}\ sec :=_{\tau_{\mathsf{v}}} * \quad //\,\textsc{AtomLocal}+\textsc{ID-VarSecret}$$
$$\Theta_0 \vdash \left[ok\colon \mathbf{s}_{\tau_{\mathsf{v}}} \vdash\!\!\dashrightarrow 1 * x \Mapsto - * z \Mapsto - * \boxed{c\mapsto [\,]} * sec{=}\mathsf{top} * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1)\right]$$
$$\mathrm{L}_2\colon \mathsf{local}\ w[8] :=_{\tau_{\mathsf{v}}} \{v\}; \quad //\,\textsc{AtomLocal} + \textsc{ID-VarArray}$$
$$\Theta_0 \vdash \left[ok\colon \mathbf{s}_{\tau_{\mathsf{v}}} \vdash\!\!\dashrightarrow 9 * x \Mapsto - * z \Mapsto - * \boxed{c\mapsto [\,]} * sec{=}\mathsf{top} * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8 * \mathop{\LARGE\ast}_{j=0}^{7}(w{+}j \Mapsto (v, \tau_{\mathsf{v}}))\right]$$
$$//\,\textsc{Frame}$$
$$\Theta_0 \vdash \left[ok\colon x \Mapsto - * z \Mapsto - * \boxed{c\mapsto [\,]} * sec{=}\mathsf{top} * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8\right] \quad //\,\textsc{EnvL}$$
$$\{[\alpha'_1]\} \vdash \left[ok\colon x \Mapsto - * z \Mapsto - * \boxed{c\mapsto [(8, \tau_{\mathsf{a}}, 0)]} * sec{=}\mathsf{top} * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8\right]$$
$$\mathrm{L}_3\colon \mathsf{recv}(c, x)_{\tau_{\mathsf{v}}}; \quad //\,(\textsc{Atom} + \textsc{ID-Recv})$$
$$\{[\alpha'_1, \alpha_1]\} \vdash \left[ok\colon x \Mapsto (8, \tau_{\mathsf{a}}, 0) * z \Mapsto - * \boxed{c\mapsto [\,]} * sec{=}\mathsf{top} * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8\right] \quad //\,\textsc{Cons}$$
$$\{[\alpha'_1, \alpha_1]\} \vdash \left[ok\colon x \Mapsto (8, \tau_{\mathsf{a}}, 0) * z \Mapsto - * \boxed{c\mapsto [\,]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8\right]$$
$$\mathsf{if}\ (x \leq 8) \quad \mathrm{L}_4\colon z :=_{\tau_{\mathsf{v}}} w[x] \quad //\,\textsc{AtomLocal}+\textsc{ID-ReadArray}$$
$$\{[\alpha'_1, \alpha_1]\} \vdash \left[ok\colon x \Mapsto (8, \tau_{\mathsf{a}}, 0) * z \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * \boxed{c\mapsto [\,]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8\right]$$
$$\mathrm{L}_5\colon \mathsf{send}(c, z)_{\tau_{\mathsf{v}}} \quad //\,\textsc{Atom}+\textsc{ID-Send}$$
$$\{[\alpha'_1, \alpha_1, \alpha_2]\} \vdash \left[ok\colon x \Mapsto (8, \tau_{\mathsf{a}}, 0) * z \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * \boxed{c\mapsto [(v_s, \tau_{\mathsf{v}}, 1)]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8\right]$$
$$//\,\textsc{EnvEr}$$
$$\Theta \vdash \left[er\colon x \Mapsto (8, \tau_{\mathsf{a}}, 0) * z \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * \boxed{c\mapsto [(v_s, \tau_{\mathsf{v}}, 1)]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * w{=}\mathsf{top}{-}8\right]$$
$$\Theta \vdash \left[\begin{array}{l} er\colon Q_v \triangleq \mathbf{s}_{\tau_{\mathsf{v}}} \vdash\!\!\dashrightarrow 9 * x \Mapsto (8, \tau_{\mathsf{a}}, 0) * z \Mapsto (v_s, \tau_{\mathsf{v}}, 1) * \boxed{c\mapsto [(v_s, \tau_{\mathsf{v}}, 1)]} * sec{=}w{+}8 * sec \Mapsto (v_s, \tau_{\mathsf{v}}, 1) \\ \qquad\qquad * w{=}\mathsf{top}{-}8 * \mathop{\LARGE\ast}_{j=0}^{7}(w{+}j \Mapsto (v, \tau_{\mathsf{v}})) \end{array}\right]$$

(c)

Figure 5 Proofs of InfDis (a), its adversary (b) and vulnerable (c) programs.

All stack variables are local and channel c is the only shared resource. As such, rely/guarantee relations describe how τ_{a} and τ_{v} transmit data over c: α_1 and α_2 capture the recv and send in τ_{v}, while α'_1 and α'_2 capture the send and recv in τ_{a}. Using AtomLocal and CASL$_{\text{ID}}$ axioms, τ_{v} executes its first two instructions. It then uses Frame to frame off unneeded resources and applies EnvL to account for τ_{a} sending $(8, \tau_{\mathsf{a}}, 0)$ over c. Using Atom with ID-Recv it receives this value in x. After using Cons to rewrite $\mathrm{sec} = \mathsf{top} * w = \mathsf{top}{-}8$ equivalently to $\mathrm{sec}{=}w{+}8 * w{=}\mathsf{top}{-}8$, it applies AtomLocal with ID-ReadArray to read from $w[x]$ (i.e. the secret value at sec$=w+8$) in z. It then sends this value over c, arriving at an error using EnvEr as the value received by the adversary τ_a is secret. The last line then adds on the resources previously framed off. The proof of τ_a in Fig. 5b is analogous.

5 Related Work

Under-Approximate Reasoning. CASL builds on and generalises CISL [18]. As with IL [16] and ISL [17], CASL is an instance of under-approximate reasoning. However, IL and ISL support only sequential programs and not concurrent ones. Vanegue [22] recently developed adversarial logic (AL) as an under-approximate technique for detecting exploits. While we model C_v and C_a as $\mathsf{C}_a \,||\, \mathsf{C}_v$ as with AL, there are several differences between AL and CASL. CASL is a general, under-approximate framework that can be 1) used to detect both exploits and bugs in concurrent programs, while AL is tailored towards exploits only; 2) *instantiated* for different classes of bugs/exploits, while the model of AL is hard-coded. Moreover, CASL borrows ideas from CISL to enable 3) *state-local* reasoning (only over parts of the state accessed), while AL supports global reasoning only (over the entire state); and 4) *thread-local* reasoning (analysing each thread in isolation), while AL accounts for all threads.

Automated Exploit Generation. Determining the exploitability of bugs is central to prioritising fixes at large scale. *Automated exploit generation* (AEG) tools craft an exploit based on predetermined heuristics and preconditioned symbolic execution of unsafe binary programs [2, 5]. Additional improvements use random walk techniques to exploit heap buffer overflow vulnerabilities reachable from known bugs [9, 1, 10]. Exploits for use-after-free vulnerabilities [23] and unsafe memory write primitives [6] have also been partially automated.

As with CASL, AEG tools are fundamentally under-approximate and may not find all attacks. Assumptions made by AEG tools are hard-coded in their implementation, in contrast to CASL which can be instantiated for new classes of vulnerabilities without redesigning the core logic from scratch. Finally, traditional AEG tools have no notion of locality and require global reasoning, making existing tools unable to cope with the path explosion problem and large targets without compromising coverage. By contrast, CASL mostly acts on local states, making it more suitable for large-scale exploit detection than current tools.

References

1 Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652, 2016.

2 Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

3 Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: Compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276514.

4 James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max Kanovich. A compositional deadlock detector for android java. In *Proceedings of ASE-36*. ACM, 2021. URL: http://www0.cs.ucl.ac.uk/staff/J.Brotherston/ASE21/deadlocks.pdf.

5 Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

6 Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1093–1110, 2020.

7 Facebook, 2021. URL: `https://fbinfer.com/`.

8 Heartbleed. The heartbleed bug, 2014. URL: `https://heartbleed.com/`.

9 Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, 2018.

10 Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1689–1706, 2019.

11 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. `doi:10.1145/69575.69577`.

12 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2676726.2676980`.

13 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. `doi:10.1145/3527325`.

14 Lars Müller. KPTI: A mitigation method against meltdown, 2018. URL: `https://www.cs.hs-rm.de/~kaiser/events/wamos2018/Slides/mueller.pdf`.

15 Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

16 Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, December 2019. `doi:10.1145/3371078`.

17 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing.

18 Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Concurrent incorrectness separation logic. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. `doi:10.1145/3498695`.

19 Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. Technical appendix, 2023. URL: `https://www.soundandcomplete.org/papers/CONCUR2023/CASL/appendix.pdf`.

20 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

21 Julien Vanegue. Zero-sized heap allocations vulnerability analysis. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–8, USA, 2010. USENIX Association.

22 Julien Vanegue. Adversarial logic. *Proc. ACM Program. Lang.*, (SAS), December 2022.

23 Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. {FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

DNN Verification, Reachability, and the Exponential Function Problem

Omri Isac
The Hebrew University of Jerusalem, Israel

Yoni Zohar
Bar-Ilan University, Ramat Gan, Israel

Clark Barrett
Stanford University, CA, USA

Guy Katz
The Hebrew University of Jerusalem, Israel

Abstract

Deep neural networks (DNNs) are increasingly being deployed to perform safety-critical tasks. The opacity of DNNs, which prevents humans from reasoning about them, presents new safety and security challenges. To address these challenges, the verification community has begun developing techniques for rigorously analyzing DNNs, with numerous verification algorithms proposed in recent years. While a significant amount of work has gone into developing these verification algorithms, little work has been devoted to rigorously studying the computability and complexity of the underlying theoretical problems. Here, we seek to contribute to the bridging of this gap. We focus on two kinds of DNNs: those that employ piecewise-linear activation functions (e.g., ReLU), and those that employ piecewise-smooth activation functions (e.g., Sigmoids). We prove the two following theorems:

(i) the decidability of verifying DNNs with a particular set of piecewise-smooth activation functions, including Sigmoid and tanh, is equivalent to a well-known, open problem formulated by Tarski; and

(ii) the DNN verification problem for any quantifier-free linear arithmetic specification can be reduced to the DNN reachability problem, whose approximation is NP-complete.

These results answer two fundamental questions about the computability and complexity of DNN verification, and the ways it is affected by the network's activation functions and error tolerance; and could help guide future efforts in developing DNN verification tools.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Formal Verification, Computability Theory, Deep Neural Networks

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.26

Related Version *Pre-print version*: `https://arxiv.org/abs/2305.06064` [26]

Funding This work was supported by the Israel Science Foundation (grant number 619/21), the Binational Science Foundation (grant numbers 2020250, 2021769, 2020704), and by the National Science Foundation (grant numbers 1814369 and 2110397).

1 Introduction

The use of artificial intelligence, and specifically that of deep neural networks (DNNs), is becoming extremely widespread – as DNNs are often able to solve complex tasks more successfully than any other computational approach. These include critical tasks in healthcare [14], autonomous driving [8], communication networks [11], and also the task of communicating with humans through text [10] – which seems to bring DNNs closer and closer to passing the famous Turing test [55].

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 26; pp. 26:1–26:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, it has been shown that even state-of-the-art DNNs are susceptible to various errors. In one infamous example, known as *adversarial perturbations*, small input perturbations that are imperceivable to the human eye are crafted to fool modern DNNs, causing them to output incorrect results selected by an attacker [21]. Adversarial perturbations thus constitute a safety and security threat, to which most DNNs are susceptible [50]. Other issues, such as privacy concerns and bias against various groups, have also been observed, making it clear that a high bar of trustworthiness must be met before stakeholders can fully accept DNNs [29].

Overcoming these weaknesses of DNNs is a significant challenge, due to their size and complexity. This is further aggravated by the fact that DNNs are machine-generated (automatically *trained* over many examples). Consequently, they are opaque to human engineers, and often fail to generalize their results to examples sufficiently different from the set of examples used for training [32]. This has sparked much interest in the verification community, which began studying verification techniques for DNNs, in order to guarantee their compliance with given specifications. In recent years, the verification community has designed and implemented multiple verification algorithms for DNNs, relying on techniques such as SMT solving [25, 28, 59], abstract interpretation [18, 22], convex relaxation [33], adversarial search [23], and many others [2,4,6,17,40,41,47,49,54,57,61]. Indeed, DNN verification technology has been making great strides recently [35].

Modern verification algorithms depend heavily on the structure of the DNN being verified, and specifically on the type of its *activation functions*. Initial efforts at DNN verification focused almost exclusively on DNNs with piecewise-linear (PWL) activation functions. It has been shown that the verification of such networks is an NP-complete problem [28, 45], and multiple algorithms have been proposed for solving it [18, 28, 31]. Although more recent approaches can handle DNNs with smooth activation functions, these algorithms are often approximate and/or incomplete [23, 36, 49]; in fact, to the best of our knowledge, there is not a single algorithm that is guaranteed to terminate with a correct answer when verifying such DNNs. This raises two important questions:

(i) does there exist a non-approximating algorithm that can *always* solve verification queries involving DNNs with non-PWL activation functions? Or, in other words, is the verification problem of DNNs with smooth and piecewise-smooth activation functions *decidable*? and

(ii) when introducing approximations, how difficult does the verification problem become with respect to the DNN, the specification, and the size of the approximation? In other words, what is the computational complexity of DNN verification with smooth and piecewise-smooth activation functions and with ϵ-error tolerance?

In this paper, we provide a partial answer for the first question, by showing that the verification problem of DNNs with smooth and piecewise-smooth activation functions is equivalent to a well-known, open problem from the field of model theory – Tarski's exponential function problem [52]. We do so by introducing a constructive bijection between verification queries of such DNNs, and instances of Tarski's open problem.

In addition, we provide a partial answer to the second question, by studying the relations between DNN verification and DNN reachability problems, and ultimately proving that they are equivalent. Even though this equivalence result was previously used [13], as far as we know, we are the first to provide a formal reduction. This enables further investigation of DNN verification with any quantifier-free linear arithmetic specification formula as a specific case of DNN reachability, without loss of generality. The latter problem is known to be NP-complete when ϵ-error tolerance is introduced in the result [44].

Formally, we prove the two following theorems:

1. The DNN verification problem for DNNs with smooth and piecewise-smooth activation functions is equivalent to Tarski's exponential function problem [52], which is a well-known open problem.
2. The DNN verification problem, with any quantifier-free linear arithmetic specification formula, can be reduced to the DNN reachability problem, which is NP-complete when some ϵ-error tolerance is allowed [44].

Our results imply a fundamental difference between the hardness of verification of DNNs with piecewise-smooth and with piecewise-linear activation functions. As far as we know, we are the first to provide any proof of this difference.

The rest of the paper is organized as follows. In Section 2 we provide background on DNNs, verification and other necessary mathematical concepts. In Section 3 and Section 4 we formally prove the two main results mentioned above. In Section 5 we discuss related work, and in Section 6 we describe our conclusions and directions for future work.

2 Background

2.1 Deep Neural Networks

Deep neural networks (DNNs) [20] are directed graphs whose nodes (neurons) are organized into layers, and whose nodes and edges are labeled with rational numbers. Nodes in the first layer, called the *input layer*, are assigned values matching the input to the DNN; and then the values of nodes in each of the subsequent layers are computed as functions of the values assigned to neurons in the preceding layer. More specifically, each node value is computed by first applying an affine transformation (linear transformation and addition of a constant) to the values from the preceding layer, and then applying a non-linear *activation function* [12] to the result. The final (output) layer, which corresponds to the output of the network, is computed without applying an activation function.

Three of the most common activation functions are the *rectified linear unit* (ReLU), which is defined as:

$$\mathrm{ReLU}(x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise;} \end{cases}$$

the *Sigmoid* function, defined as:

$$\sigma(x) : \mathbb{R} \to (0,1) = \frac{\exp(x)}{\exp(x)+1}$$

where exp is the exponential function; and the *hyperbolic tangent*, defined as:

$$\tanh(x) : \mathbb{R} \to (-1,1) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

The latter two activation functions are both injective, and their inverses are defined as follows:

$$\forall x \in (0,1) : \sigma^{-1}(x) = \ln(\frac{x}{1-x})$$
$$\forall x \in (-1,1) : \tanh^{-1}(x) = \frac{1}{2}\ln(\frac{1+x}{1-x})$$

where ln is the natural logarithm function. In addition, we consider the NLReLU activation function [12, 30], denoted τ for short, which is defined as:

$$\text{NLReLU}(x) = \tau(x) = \ln(\text{ReLU}(x) + 1)$$

A simple DNN with four layers appears in Figure 1, where all biases are set to zero and are ignored. For input $\langle 1, 2, 1\rangle$, the first node in the second layer evaluates to $\text{ReLU}(1 \cdot 1 + 2 \cdot (-1)) = \text{ReLU}(-1) = 0$; the second node in the second layer evaluates to $\text{ReLU}(2 \cdot 1 + 1 \cdot (-1)) = \text{ReLU}(1) = 1$; and the node in the third layer evaluates to $\sigma(0 - 1) = \sigma(-1)$. Thus, the node in the fourth (output) layer evaluates to $4 \cdot \sigma(-1)$.

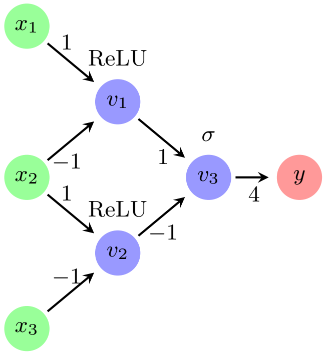


Figure 1 A toy DNN.

Formally, a DNN $\mathcal{N} : \mathbb{R}^m \to \mathbb{R}^k$ is a sequence of n layers $L_0, ..., L_{n-1}$ where each layer L_i consists of $s_i \in \mathbb{N}$ nodes, denoted $v_i^1, ..., v_i^{s_i}$, and biases $p_i^j \in \mathbb{Q}$ for each v_i^j. Each directed edge in the DNN is of the form (v_{i-1}^l, v_i^j) and is labeled with $w_{i,j,l} \in \mathbb{Q}$. The assignment to the nodes in the input layer is defined by $v_0^j = x_j$, where $\overline{x} \in \mathbb{R}^m$ is the input vector, and the assignment for the j^{th} node in the $1 \leq i < n - 1$ layer is computed as

$$v_i^j = f_i^j \left(\sum_{l=1}^{s_{i-1}} w_{i,j,l} \cdot v_{i-1}^l + p_i^j \right)$$

for some activation function $f_i^j : \mathbb{R} \to \mathbb{R}$. Finally, neurons in the output layer are computed as:

$$v_{n-1}^j = \sum_{l=1}^{s_{n-2}} w_{n-1,j,l} \cdot v_{n-2}^l + p_{n-1}^j$$

where $w_{i,j,l}$ and p_i^j are (respectively) the predetermined weights and biases of $\mathcal{N}$. The size of a network $|\mathcal{N}|$ is defined as the overall number of its neurons.

2.2 Formal Analysis of DNNs

The formal methods community has tackled the formal analysis of DNNs primarily along two axes: DNN *verification*, and DNN *reachability*. These are two related formulations, as reachability problems may be expressed as verification problems in a straightforward manner. In this paper, we further study the connections between these two formulations.

DNN Verification. Let $\mathcal{N} : \mathbb{R}^m \to \mathbb{R}^k$ be a DNN and $P : \mathbb{R}^{m+k} \to \{\top, \bot\}$ be a property, where $\top, \bot$ represent the values for which the property does and does not hold, respectively. The *DNN verification problem* is to decide whether there exist $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^k$ such that $(\mathcal{N}(x) = y) \wedge P(x, y)$ holds. In particular, a verification query is always expressed as an existential formula. If such x and y exist, we say that the verification query $\langle \mathcal{N}, P \rangle$ is *satisfiable* (SAT); and otherwise, we say that it is *unsatisfiable* (UNSAT).

A verification algorithm is *sound* if it does not return UNSAT for satisfiable queries, and does not return SAT for unsatisfiable queries (in other words, if its answers are always correct); and is *complete* if it always terminates, for any query.

So far, there have been several efforts at studying the complexity-theoretical aspects of DNN verification [15, 24, 27, 28, 44–46]. Most previous work was focused on DNNs with piecewise-linear activation function (specifically, ReLUs), while leaving many open questions about the complexity-theoretical aspects of verifying DNNs with smooth and piecewise-smooth activation functions.

DNN Reachability. Given a DNN $\mathcal{N} : \mathbb{R}^m \to \mathbb{R}$, a function $o : \mathbb{R} \to \mathbb{R}$ and an input set $\mathcal{X} \subseteq [0, 1]^m$, the *DNN reachability problem* is to compute $\sup\limits_{x \in \mathcal{X}} o(\mathcal{N}(x))$ and $\inf\limits_{x \in \mathcal{X}} o(\mathcal{N}(x))$, perhaps up to some ϵ-error tolerance. For DNNs with Lipschitz-continuous activation functions, either smooth or piecewise-linear, (such as σ and ReLU), the reachability problem with some ϵ-error tolerance is NP-complete, in the size of the network and ϵ [44]. In this work, we consider a decision version for the problem where o is the identity, deciding whether $\underset{x \in \mathcal{X}}{sup}\ \mathcal{N}(x) \geq 0$ is achieved for some $x \in \mathcal{X}$. This decision version with some ϵ-error tolerance is then to decide whether $\underset{x \in \mathcal{X}}{sup}\ \mathcal{N}(x) \geq -\epsilon$ is achieved for some $x \in \mathcal{X}$. In addition, we may assume that the input of $\mathcal{N}$ is within $[0, 1]^m$, as the input domain may be normalized before the network is evaluated.

For example, consider the DNN depicted in Figure 1. A possible *verification* query for this DNN is given by a property P that returns $\top$ if and only if $(x_1, x_2, x_3) \in [0, 1]^3 \wedge (y \in [0.5, 0.75] \vee y \in [0, 0.25])$; i.e., if there exists an input in the $[0, 1]^3$ cube, for which $y \in [0.5, 0.75] \vee y \in [0, 0.25]$. A possible *reachability* query is to check whether there exists an input in the domain $[0, 1] \times [0, 0.5] \times [0.5, 1]$, for which $y \geq 0$. This reachability query can trivially be represented as a verification property P', which returns $\top$ if and only if $(x_1, x_2, x_3) \in [0, 1] \times [0, 0.5] \times [0.5, 1] \wedge y \geq 0$. For any $\epsilon > 0$, the equivalent reachability query with ϵ tolerance is to decide if there exists an input in the domain $[0, 1] \times [0, 0.5] \times [0.5, 1]$, for which $y \geq -\epsilon$.

2.3 Decidability and Mathematical Logic

Mathematical Logic. In mathematical logic, a *signature* Σ is a set of symbols, representing functions and relations. A Σ*-formula* is a formula, comprised of atoms and relations that appear in Σ, the usual logical operators $(\wedge, \neg, \vee, \rightarrow, \leftrightarrow)$, and the quantifiers $\forall$ (universal) and $\exists$ (existential). A variable affixed with a quantification symbol is a *bounded variable*; and otherwise it is a *free variable*. A formula without free variables is called a *sentence*, and a formula without bounded variables is called a *quantifier-free formula*. A formula with variables $\overline{x} = (x_1, ..., x_n)$ of the form $\exists(\overline{x}).\varphi$ where φ is quantifier-free is called an *existential formula*. A Σ*-theory* is a set of Σ-sentences. A Σ*-model* $\mathcal{M}$ is comprised of a set of elements, denoted $|\mathcal{M}|$, and an *interpretation* for all Σ functions and relations; that is, a definition $f^{\mathcal{M}} : |\mathcal{M}|^n \to |\mathcal{M}|$ for every n-ary function $f \in \Sigma$, and a definition $r^{\mathcal{M}} \subseteq |\mathcal{M}|^m$ or every m-ary relation $r \in \Sigma$. If the interpretation of a Σ-sentence φ is true within a model $\mathcal{M}$, we say that $\mathcal{M}$ *satisfies* φ, and denote $\mathcal{M} \models \varphi$. If $\mathcal{M}$ satisfies all sentences in a Σ-theory $\mathcal{T}$, then $\mathcal{M}$ is a $\mathcal{T}$*-model*, denoted $\mathcal{M} \models \mathcal{T}$. Given some model $\mathcal{M}$ over signature Σ, we define the theory $Th(\mathcal{M})$ as the set of all Σ-sentences φ such that $\mathcal{M} \models \varphi$. It is then trivial that $\mathcal{M} \models Th(\mathcal{M})$.

For example, let Σ be the set $\{+, -, \cdot, 0, 1, <\}$ where $+, -, \cdot$ are 2-ary functions, $0, 1$ are 0-ary functions and $<$ is a 2-ary relation. Let $\mathcal{M}$ be a model defined over $\mathbb{R}$ with addition, subtraction, multiplication, the constants 0,1 and the usual order. Then $Th(\mathcal{M})$ is the set of all Σ-sentences that $\mathcal{M}$ satisfies, such as $\forall x : x \cdot 0 = 0$.

Theory Decidability. For a Σ-theory $\mathcal{T}$ and a Σ-sentence φ, we say that φ is $\mathcal{T}$*-valid* and denote $\mathcal{T} \vdash \varphi$, if every model of $\mathcal{T}$ satisfies φ. Furthermore, we say that φ is $\mathcal{T}$*-satisfiable* if there exists a model $\mathcal{M}$ of $\mathcal{T}$, for which $\mathcal{M} \models \varphi$; and that φ is $\mathcal{T}$*-unsatisfiable* if $\mathcal{M} \not\models \varphi$ for all models $\mathcal{M}$ of $\mathcal{T}$. Satisfiability and validity are closely connected, as φ is $\mathcal{T}$-valid if and only if $\neg\varphi$ is $\mathcal{T}$-unsatisfiable. A theory $\mathcal{T}$ is *decidable* if there exists an algorithm that, for any sentence φ, decides whether $\mathcal{T} \vdash \varphi$, within a finite number of steps. If φ is valid, the algorithm returns $\top$; and otherwise, it returns $\bot$. Due to the connection of satisfiability and validity, validity-checking algorithms may also be used to decide satisfiability, and vice versa. In particular, for any theory $Th(\mathcal{M})$ for some model $\mathcal{M}$, all $Th(\mathcal{M})$-models satisfy exactly the same sentences, so validity and satisfiability are equivalent. Thus, throughout this paper we use decision procedures to decide the $Th(\mathcal{M})$-satisfiability of formulas. In addition, when considering *quantifier-free* formulas (i.e., a formula where all variables are free), all of the formula's variables are implicitly existentially quantified. In this case, for any quantifier-free formula $\varphi(\overline{x})$ with variable vector $\overline{x}$, the satisfiability problem of $\varphi(\overline{x})$ with respect to a model $\mathcal{M}$ is equivalent to deciding whether $\mathcal{M} \models \exists\overline{x}.\varphi(\overline{x})$. Similarly, the satisfiability problem of $\varphi(\overline{x})$ with respect to a theory $\mathcal{T}$ is equivalent to deciding whether $\exists\overline{x}.\varphi(\overline{x})$ is $\mathcal{T}$-satisfiable.

It has previously been shown that the theory of the real field $Th(\mathbb{R}, +, -, \cdot, 0, 1, <)$ is decidable [51], and that the theory of the real field with the transcendental functions $\exp, \sin$ and the constants $\log 2, \pi$ is undecidable [42]. The question of the decidability of $Th(\mathbb{R}, +, -, \cdot, 0, 1, <, \exp)$ has remained an open problem since the 1950's [52], and it is commonly known as Tarksi's *exponential function problem.*

A theory $\mathcal{T}$ is *stably-infinite* if for every quantifier-free formula φ, the satisfiability of φ in $\mathcal{T}$ implies that φ is satisfiable in some infinite model of $\mathcal{T}$. It is then immediate that for any infinite model $\mathcal{M}$, $Th(\mathcal{M})$ is stably-infinite.

Given two decidable, stably-infinite theories $\mathcal{T}_1$ and $\mathcal{T}_2$ defined over disjoint sets of symbols, for any quantifier-free formulas $F_1 \in \mathcal{T}_1, F_2 \in \mathcal{T}_2$, the formula $F_1 \wedge F_2$ is decidable as well [38]. The *Nelson-Oppen method* [38] is a well-known method for combining two decision procedures for two theories into a decision procedure for the quantifier-free fragment of their union.

Equisatisfiability. Two formulas φ and ψ are *equisatisfiable* if φ is satisfiable if and only if ψ is satisfiable. For example, the formulas $\varphi := (a + b) * (a - b) = 0$ and $\psi := c * d = 0 \wedge c = a + b \wedge d = a - b$ are equisatisfiable. Note that φ and ψ may be formulated in different theories, $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. In this case, we say that the formulas are equisatisfiable if φ is $\mathcal{T}_1$-satisfiable if and only if ψ is $\mathcal{T}_2$-satisfiable.

Function Definability. For any signature Σ and an n-ary function f, not necessarily in Σ, we say that f is *definable* in a Σ-model $\mathcal{M}$ if there exists a Σ-formula $\psi(x_1, ..., x_n, y, z_1, ..., z_m)$ over the variables $x_1, ..., x_n, y$ such that for any elements $a_1, ..., a_n, b$ in $\mathcal{M}$ we have that $\mathcal{M} \models \exists z_1, ..., z_m.\psi(a_1, ..., a_n, b, z_1, ..., z_m)$ if and only if $b = f(a_1, ..., a_n)$. We say that f is definable in a Σ-theory $\mathcal{T}$ if it is definable in all models of $\mathcal{T}$.

Model-Completeness. In model theory, the concept of model-completeness has several equivalent definitions. For our purposes, a theory $\mathcal{T}$ is model-complete if and only if any formula in the theory has an equivalent existential formula (modulo $\mathcal{T}$). This means that the existential formulas in $\mathcal{T}$ can express all the formulas in it. $Th(\mathbb{R}, +, -, \cdot, 0, 1, <, \exp)$ is known to be model-complete [58].

3 Decidability of DNN Verification

In this section, we prove our first main result: the decidability of verifying a DNN with the activation functions ReLU, σ, tanh and τ is equivalent to the decidability of $Th(\mathbb{R}, +, -, \cdot, 0, 1, <, \exp)$. The decidability of this theory is an open problem [52]. Thus, the equivalence implies that the decidability of DNN verification for DNNs with the activation functions ReLU, σ, tanh and τ is an open problem as well.

For simplicity, we denote $\mathcal{T}_{\mathbb{R}} = Th(\mathbb{R}, +, -, \cdot, 0, 1, <)$, $\mathcal{T}_{\exp} = Th(\mathbb{R}, +, -, \cdot, 0, 1, <, \exp)$, and $\mathcal{T}_{\sigma} = Th(\mathbb{R}, +, -, \cdot_{q\in\mathbb{Q}}, 0, 1, <, \sigma, \tanh, \tau)$, where $\cdot_q$ is an unary function, interpreted as the multiplication with a constant $q \in \mathbb{Q}$. We use $\Sigma_\sigma, \Sigma_{\exp}$ and $\Sigma_{\mathbb{R}}$ to denote the signatures of $\mathcal{T}_\sigma, \mathcal{T}_{\exp}$ and $\mathcal{T}_{\mathbb{R}}$, respectively. Note that for any DNN, weights and biases are in $\mathbb{Q}$, and can thus be expressed as $\mathcal{T}_{\mathbb{R}}$-terms. Therefore, we can express the affine constraints of the network as $\mathcal{T}_{\mathbb{R}}$-formulas. In addition, any constraint of the form $f = \text{ReLU}(b)$ can be expressed as the formula $(f = b \leftrightarrow b > 0) \wedge (f = 0 \leftrightarrow b \leq 0)$, and thus ReLU is definable in $\mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\exp}$ and $\mathcal{T}_\sigma$. Therefore, without loss of generality, we need not add a function symbol to Σ_σ to express DNNs with ReLU activation functions (or any other piecewise-linear function). Our goal is then to show that the decidability of $\mathcal{T}_{\exp}$ is equivalent to the decidability of all existential formulas of $\mathcal{T}_\sigma$.

▶ **Example.** We begin with an example that illustrates this equivalence. For the first direction, consider the toy DNN depicted in Figure 1, and let $x_1, x_2, x_3, b_1, f_1, b_2, f_2, b_3, f_3$, and y be the variables of the network. Variables x_1, x_2, x_3 represent the input variables, variables $b_1, f_1, b_2, f_2, b_3, f_3$ represent the inputs and outputs of nodes v_1, v_2, v_3, respectively, and variable y represents the network's output. Let P be the property restricting the input to be within $[0, 1]^3$ and the output to be in $[1, 2]$. The verification query for the network in Figure 1 and P is then:

$$\begin{gathered}
\bigwedge_{i\in 1,2,3} [(x_i \geq 0) \wedge (x_i \leq 1)] \wedge \\
(x_1 - x_2 = b_1) \wedge (x_2 - x_3 = b_2) \wedge \\
\bigwedge_{i\in 1,2} [(f_i = b_i) \leftrightarrow (b_i > 0)] \wedge [(f_i = 0) \leftrightarrow (b_i \leq 0)] \wedge \\
(f_1 - f_2 = b_3) \wedge (f_3 = \sigma(b_3)) \wedge (4 \cdot f_3 = y) \wedge \\
(1 \leq y) \wedge (y \leq 2)
\end{gathered}$$

This is a $\mathcal{T}_\sigma$ query, which can be expressed as a query in $\mathcal{T}_{\exp}$, since $\sigma(x) = \frac{\exp(x)}{1+\exp(x)}$. The equivalent $\mathcal{T}_{\exp}$ query is:

$$\begin{gathered}
\bigwedge_{i\in 1,2,3} [(x_i \geq 0) \wedge (x_i \leq 1)] \wedge \\
(x_1 - x_2 = b_1) \wedge (x_2 - x_3 = b_2) \wedge \\
\bigwedge_{i\in 1,2} [(f_i = b_i) \leftrightarrow (b_i > 0)] \wedge [(f_i = 0) \leftrightarrow (b_i \leq 0)] \wedge \\
(f_1 - f_2 = b_3) \wedge [(\exp(b_3) + 1) \cdot f_3 = \\
\exp(b_3)] \wedge (4 \cdot f_3 = y) \\
(1 \leq y) \wedge (y \leq 2)
\end{gathered}$$

For the second direction, we begin by demonstrating a purification process of a given formula $\varphi := \exp(a+b) = \exp(a) \cdot \exp(b)$ into a formula in $\mathcal{T}_\sigma$. We assume that we can define $\psi_{c=a\cdot b}$ and $\psi_{y=\exp(x)}$ in Σ_σ, which are defined over the variables a, b, c and x, y, respectively, and that witness the definability of the functions $\cdot$ and exp in $\mathcal{T}_\sigma$. Therefore, the formula

$$\psi_{p=\exp(a)} \wedge \psi_{q=\exp(b)} \wedge \psi_{r=\exp(a+b)} \wedge \psi_{r=p\cdot q}$$

is equisatisfiable to $\exp(a+b) = \exp(a) \cdot \exp(b)$.

Formally, we prove the following theorem:

▶ **Theorem 1.** *The decidability of verifying DNNs with $\sigma, \tanh, \tau$ and ReLU activation functions is equivalent to the decidability of $Th(\mathbb{R}, +, -, \cdot, 0, 1, <, \exp)$.*

Proof. The first direction of the proof is similar to a technique proposed by Ivanov et al. [27]. Assume there exists a decision procedure for $\mathcal{T}_{\exp}$, and let $F \in \Sigma_\sigma$ be a DNN verification query. For any appearance of $\sigma(t)$ for some term t, we replace $t, \sigma(t)$ with the fresh variables x, y, respectively and add the conjunction:

$$(\exp(x) + 1) \cdot y = \exp(x) \wedge x = t$$

to the resulting formula. This is done in a way similar to the one described in the example. Similarly, for any appearance of $\tanh(t)$ for some term t, we replace $t, \tanh(t)$ with the fresh variables x, y, respectively and add the conjunction:

$$(\exp(x) + \exp(-x)) \cdot y = \exp(x) - \exp(-x) \wedge x = t$$

to the resulting formula. For defining τ, we first define:

$$\psi_{f=\mathrm{ReLU}(b)} := (f = b \leftrightarrow b > 0) \wedge (f = 0 \leftrightarrow b \leq 0)$$

Now, for any appearance of $\tau(t) = \ln(\mathrm{ReLU}(t) + 1)$ for some term t, we replace $t, \tau(t)$ with the fresh variables x, y, respectively and add the conjunction:

$$\psi_{z=\mathrm{ReLU}(x)} \wedge \exp(y) = z + 1 \wedge x = t$$

to the resulting formula, where z is an additional fresh variable. After repeating this process iteratively, we convert any $F \in \Sigma_\sigma$ to an equisatisfiable formula $F' \in \Sigma_{\exp}$. We then use the decision procedure to decide the satisfiability of F'.

The second direction of the proof is more complex. Assume we have a sound and complete verification procedure for DNNs with $\sigma, \tanh$ and τ activation functions; that is, a decision procedure for deciding the satisfiability of quantifier-free Σ_σ-formulas in $\mathcal{T}_\sigma$.

Since $\mathcal{T}_{exp}$ is model-complete, it is tempting to try and construct a decision procedure for the existential formulas of $\mathcal{T}_{\exp}$. However, to the best of our knowledge, given a general formula in $\mathcal{T}_{\exp}$ it is not known how to effectively derive its equivalent existential formula. In order to circumvent this issue, we consider instead a fourth theory, $\mathcal{T}_e = Th(\mathbb{R}, +, -, \cdot, 0, 1, <, e)$, defined over the signature Σ_e, where $e : \mathbb{R} \to \mathbb{R}$ with $e(x) = \exp(\frac{1}{1+x^2})$ is the *restricted* exponential function. It has been shown by Macintyre and Wilkie [34] that the decidability of this theory implies the decidability of $\mathcal{T}_{exp}$, and that given any formula in the language of $\mathcal{T}_e$, one can effectively find an equivalent existential formula (in $\mathcal{T}_e$). Therefore, it is enough for our purpose to consider any existential formula $\exists \overline{x}.\varphi \in \Sigma_e$, and decide the satisfiability of φ in $\mathcal{T}_e$.

Let $\exists \overline{x}.\varphi \in \Sigma_e$ be an existential formula, where φ is a quantifier-free formula. We construct a Σ_σ-formula ψ, equisatisfiable to φ. In this construction, all variables are implicitly existentially quantified. In order to do so, it is enough to define formulas $\psi_{c=a\cdot b}$ and $\psi_{y=e(x)}$ over the variables a, b, c and x, y respectively, and witness the definability of the functions $\cdot$ and e in $\mathcal{T}_\sigma$. In this case, given any formula $\varphi \in \Sigma_e$, we can iteratively replace any occurrence of terms of the form $t \cdot s$ with the fresh variable p and add the conjunction $\psi_{p=t\cdot s}$, and occurrences of terms of the form $e(x)$ with the fresh variable q and add the conjunction $\psi_{q=e(x)}$. This process terminates with a Σ_σ-formula ψ equisatisfiable to φ, allowing us to apply the decision procedure to ψ.

To complete the proof, it remains to show how $\psi_{c=a\cdot b}$ and $\psi_{y=e(x)}$ can be defined using the formula $\psi_{y=ln(x)}$. We show the construction of $\psi_{y=ln(x)}$ later, in Lemma 2, and we use it here to define both $\psi_{c=a\cdot b}$ and $\psi_{y=e(x)}$.

We start by defining $\psi_{c=a\cdot b}$. Note that $\forall a, b > 0$, it holds that $\ln(a \cdot b) = \ln(a) + \ln(b)$; and so $a \cdot b = \exp(\ln(a) + \ln(b))$, assuming $a, b > 0$. This equality can be expressed using the formula:

$$\theta_{c=a\cdot b} := \psi_{p=\ln(a)} \wedge \psi_{q=\ln(b)} \wedge \psi_{p+q=\ln(c)},$$

where c represents the value of $a \cdot b$, and p, q are fresh variables. For defining $\psi_{c=a\cdot b}$ for all $a, b \in \mathbb{R}$, we split into cases, and write:

$$\begin{aligned}&\psi_{c=a\cdot b} := [(a > 0 \wedge b > 0) \rightarrow \theta_{c=a\cdot b}]\wedge\\ &[(a < 0 \wedge b > 0) \rightarrow \theta_{-c=-a\cdot b}]\wedge\\ &[(a > 0 \wedge b < 0) \rightarrow \theta_{-c=a\cdot -b}]\wedge\\ &[(a < 0 \wedge b < 0) \rightarrow \theta_{c=-a\cdot -b}]\wedge\\ &[(a = 0 \vee b = 0) \leftrightarrow c = 0],\end{aligned}$$

which represents the function $\cdot$ and witnesses its definability in $\mathcal{T}_\sigma$.

We now define $\psi_{y=e(x)}$. Recall that $e(x) = \exp(\frac{1}{x^2+1})$, so in order to define $\psi_{y=e(x)}$ we use both $\psi_{c=a\cdot b}$ and $\psi_{y=\ln(x)}$:

$$\psi_{y=e(x)} := \psi_{a=\ln(y)} \wedge \psi_{1=a\cdot(b+1)} \wedge \psi_{b=x\cdot x}$$

where a, b are fresh variables.

We have defined both $\psi_{c=a\cdot b}$ and $\psi_{y=e(x)}$, which concludes our proof. ◀

For the completeness of this section, we provide now the proof of Lemma 2, which shows the construction of $\psi_{y=ln(x)}$:

▶ **Lemma 2.** *The natural logarithm function* ln *is definable in* $\mathcal{T}_\sigma$.

Proof. First, observe that for any $x \geq 1$ we have that $\tau(x-1) = \ln(\text{ReLU}(x-1)+1) = \ln(x-1+1) = \ln(x)$. Second, observe that $\forall x \in (0,1)$, the inverses of σ and tanh are defined and are equal to:

$$\sigma^{-1}(x) = \ln(\frac{x}{1-x}) = \ln(x) - \ln(1-x)$$

and

$$\tanh^{-1}(x) = \frac{1}{2}\ln(\frac{1+x}{1-x}) = \frac{1}{2}(\ln(1+x) - \ln(1-x))$$

We conclude that:

$$\forall x \in (0,1) : \sigma^{-1}(x) - 2\tanh^{-1}(x) + \tau(x) = \ln(x) - \ln(1-x) - \ln(1+x) + \ln(1-x) + \ln(x+1) = \ln(x)$$

We can express this relation using the following formula, and the fresh variables a, b, c:

$$\theta_{x,y} := [x = \sigma(a)] \wedge [x = \tanh(b)] \wedge [c = \tau(x)] \wedge [y = a - 2b + c]$$

Where $2b$ is syntactic sugar for $\cdot_2(b)$. Thus, we can define:

$$\psi_{y=\ln(x)} := [(1 < x) \to (y = \tau(x-1))] \wedge [(x = 1) \leftrightarrow (y = 0)] \wedge [(0 < x < 1) \to \theta_{x,y}] \wedge [0 < x]$$

which concludes the proof. ◀

4 DNN Verification is DNN Reachability

The two main formal analysis approaches for DNNs, verification and reachability, are closely connected: a DNN reachability instance can be formulated as DNN verification in a straightforward manner, as in the example in Section 2.2. Presently, DNN analysis algorithms and tools typically support one of the two formulations. Here, we prove that DNN verification and DNN reachability are in fact equivalent.

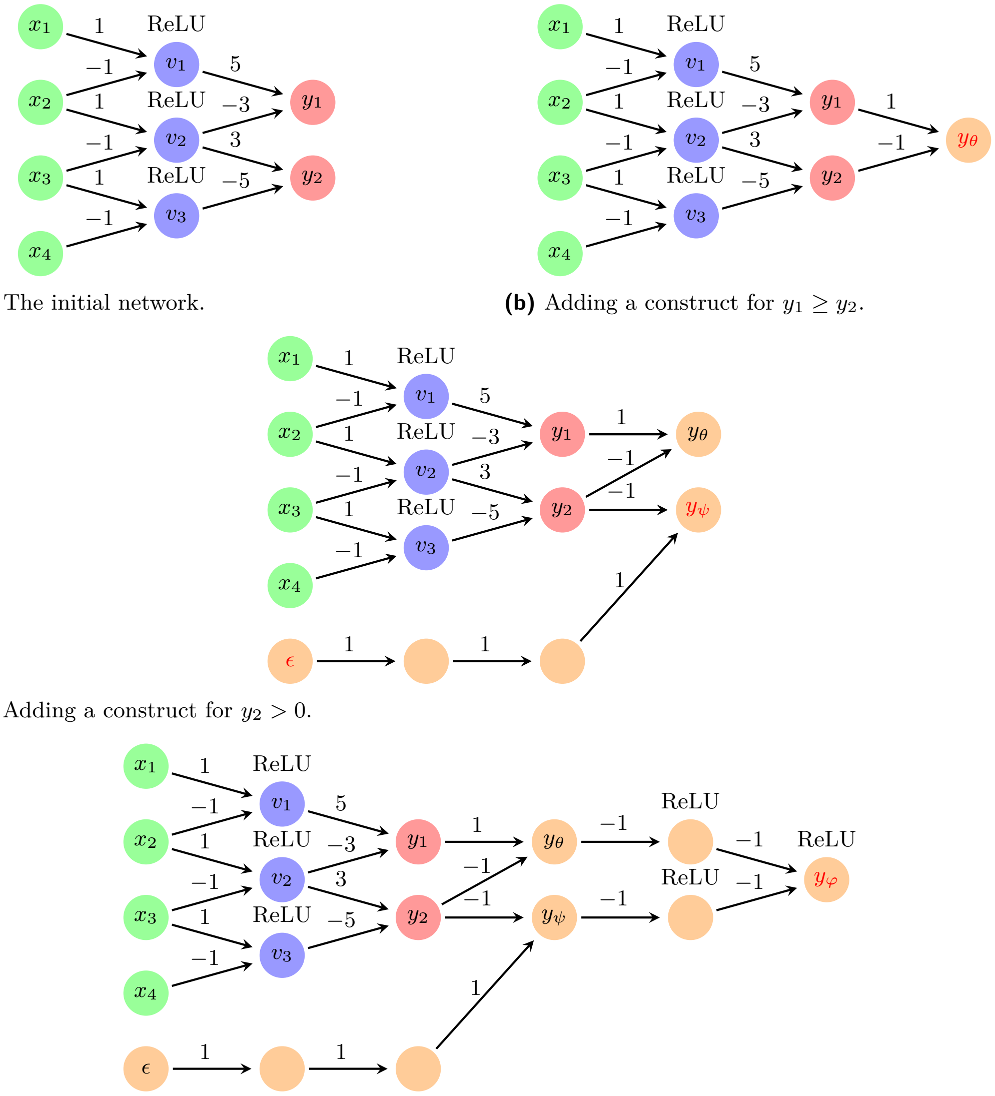
In this part, we consider DNNs that use both piecewise-linear and Sigmoidal activation functions. We formally prove that any instance of the DNN verification problem, with any specification expressible by a quantifier-free linear arithmetic formula, can be reduced to an instance of the DNN reachability problem. Since the reachability problem is a specific case of verification, we ultimately prove that for DNNs, reachability and verification are equivalent. Since it was shown that approximation of DNN reachability queries with Lipschitz-continuous activation functions (such as σ and ReLU) is NP-complete [44], we deduce that the DNN verification problem is reducible to a problem whose approximation is NP-complete. The reduction involves adding an additional input, denoted ϵ, and we use (x, ϵ) to denote the concatenation of ϵ to the input vector x. Formally, we prove the following theorem:

▶ **Theorem 3.** *Let $\mathcal{N} : \mathbb{R}^m \to \mathbb{R}^k$ be a neural network, let φ be a quantifier-free property with atoms expressing affine constraints over variables y_i of $\mathcal{N}$, and let $X \subseteq \mathbb{R}^m$. There exists a neural network $\mathcal{N}' : \mathbb{R}^{m+1} \to \mathbb{R}$, with $|\mathcal{N}'| = O(|\mathcal{N}| + |\varphi|)$ such that the two following conditions are equivalent:*

- $\exists x \in X.\quad \mathcal{N}(x) \models \varphi$
- $\exists (x, \epsilon) \in X \times (0, 1]\,.\quad \mathcal{N}'(x, \epsilon) \geq 0$

▶ **Example.** We begin with an example for constructing $\mathcal{N}'$, given some DNN $\mathcal{N}$, and a property φ. Consider first $\mathcal{N} : \mathbb{R}^4 \to \mathbb{R}^2$ as depicted in Figure 2a and $\varphi := (y_1 > 0) \wedge (y_1 \geq y_2)$. We denote $\theta := y_1 \geq y_2$ and $\psi := y_1 > 0 \equiv \neg(-y_1 \geq 0)$. In Figure 2 we start with the initial DNN $\mathcal{N}$, and then iteratively add new nodes to $\mathcal{N}$. In particular, we show how to add neurons that are active if and only if $\mathcal{N} \models \theta$ and $\mathcal{N} \models \psi$, respectively in Figure 2b and Figure 2c. Lastly, in Figure 2d we show how to add the output neuron, such that $\exists x \in X.\quad \mathcal{N}(x) \models \varphi$ if and only if $\exists (x, \epsilon) \in X \times (0, 1]\,.\quad \mathcal{N}'(x, \epsilon) \geq 0$. This concludes our example.

We now prove the theorem by induction on the generating sequence of φ; that is, a sequence of sub-formulas of φ: $\varphi_1, ..., \varphi_n$ such that $\forall i, j$ if $j > i$ then φ_j cannot be a sub-formula of φ_i, and $\varphi_n = \varphi$. This allows inductive proofs over the formulas [48]. For example, a generating sequence for the formula



(a) The initial network.

(b) Adding a construct for $y_1 \geq y_2$.

(c) Adding a construct for $y_2 > 0$.

(d) Adding a construct for φ, as a conjunction.

Figure 2 Construction of a reachability problem for $\mathcal{N} \models \varphi$.

$$\varphi := \exists x, y.(3x \geq 7) \wedge \neg(y \geq x)$$

is:

$$y \geq x, 3x \geq 7, \neg(y \geq x), (3x \geq 7) \wedge \neg(y \geq x), \exists x, y.(3x \geq 7) \wedge \neg(y \geq x)$$

Proof. Without loss of generality, assume that φ is composed of atoms, negations, and conjunctions. In addition, assume that each variable y_j is an output variable (otherwise, we may add neurons with the identity as activation function from the neuron outputting y_j to the output layer). For every step i in the generating sequence $\varphi_1, ..., \varphi_k = \varphi$, we add a constant number of output neurons, such that for any $x \in \mathbb{R}^m$, the resulting DNN $\mathcal{N}'_i$, satisfies $\mathcal{N}'_i \geq 0$ (for the last constructed output neuron) if and only if $\mathcal{N}(x) \models \varphi_i$. Below we explain the construction and prove its correctness; and in Figure 3 we show its visual representation.

Base Cases.

1. Let $\varphi := \top$. In this case, $\mathcal{N}'$ is constructed from $\mathcal{N}$ by adding a single affine neuron with no activation function, and with its input edges with weight 0 from all output nodes of $\mathcal{N}$. This also maintains the convention that the output layer does not have an activation function. Therefore, $\forall x \in \mathbb{R}^m : \mathcal{N}'(x, \epsilon) \geq 0$ if and only if $\sum_i 0 \geq 0$, which is equivalent to $\top$. The case of $\bot$ is covered by our handling of negations.
2. Let $\varphi := \sum_i c_i \cdot y_i + b \geq 0$. In this case, $\mathcal{N}'$ is constructed from $\mathcal{N}$ by adding a single affine neuron with no activation function, with its input edges with weight c_i from every output neuron y_i of $\mathcal{N}$, and a bias b. Therefore, $\forall x \in \mathbb{R}^m$ and $\mathcal{N}(x) = y$ we have that $\sum_i c_i \cdot y_i + b \geq 0$ if and only if $\mathcal{N}'(x, \epsilon) \geq 0$. We note that equality can be handled using conjunctions, while strict inequalities can be handled using negations.

Inductive step.

1. Let $\varphi := \psi \wedge \theta$, and let y_ψ, y_θ be the values of the neurons such that $y_\psi \geq 0, y_\theta \geq 0$ if and only if $\mathcal{N}(x) \models \psi, \theta$, respectively. Consider:

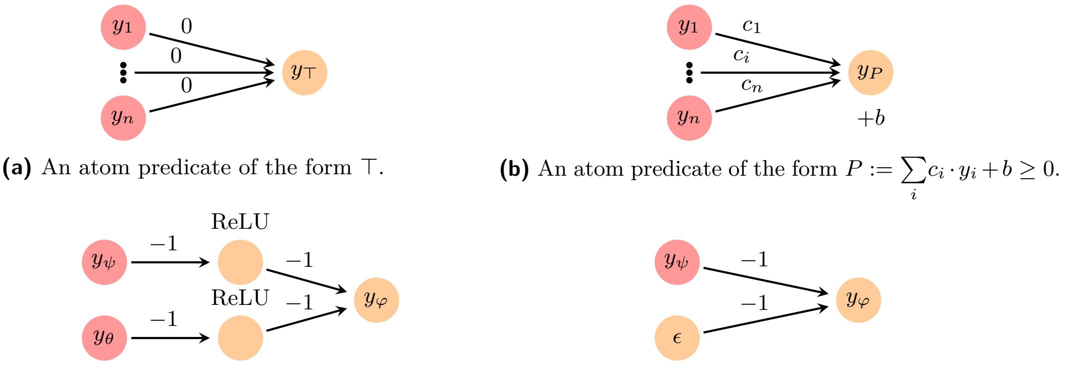
   $$y_\varphi = -\mathrm{ReLU}(-y_\psi) - \mathrm{ReLU}(-y_\theta)$$

   In this case, we have that $y_\varphi \geq 0$ if and only if $y_\psi \geq 0 \wedge y_\theta \geq 0$. We can see this since if $y_\psi \geq 0 \wedge y_\theta \geq 0$ then both $-\mathrm{ReLU}(-y_\psi)$ and $-\mathrm{ReLU}(-y_\theta)$ equal zero. Otherwise, at least one of $-\mathrm{ReLU}(-y_\psi)$ and $-\mathrm{ReLU}(-y_\theta)$ is negative (and the other is non-positive). Thus, we add two ReLU neurons, with a single -1 input edge from each of the nodes corresponding to y_ψ, y_θ, respectively. We then add a third neuron with two -1 edges from the ReLU nodes and no activation function.
2. Let $\varphi := \neg\psi$, and let y_ψ be the value of the neuron such that $y_\psi \geq 0$ if and only if $\mathcal{N}(x) \models \psi$. In this case, we first need to add a new ϵ_φ input neuron, and restrict it to $\epsilon_\varphi > 0$. Then, observe that $\neg(y_\psi \geq 0) \equiv y_\psi < 0$ if and only if there exists some $\epsilon > 0$ s.t. $\epsilon + y_\psi \leq 0$, or equivalently $-\epsilon - y_\psi \geq 0$. Therefore, we add a new neuron with no activation function, with a skip connection from the ϵ_φ neuron with weight -1, and with a -1 weight from y_ψ, resulting in $y_\varphi = -\epsilon_\varphi - y_\psi$. We note that since there are finitely many such constructions, we can choose the minimal ϵ implied by all of them, and again choose the minimum of it and 1. Thus, a single $\epsilon \in (0, 1]$ suffices. In addition, the use of the skip connections can be replaced with a line of ReLU neurons, starting with the ϵ neuron and feed-forwarding to a neuron on every layer. This construction does not affect the asymptotic size of $\mathcal{N}'$.

On every step of the recursion, we added a constant number of neurons to the network, such that $\forall x \in \mathbb{R}^m$, $\mathcal{N}'(x, \epsilon) \geq 0$ if and only if $\mathcal{N}(x) \models \varphi_i$. This concludes our proof. ◀

5 Related Work

The complexity of DNN verification has been studied mainly for DNNs with piecewise-linear activation functions – specifically, the ReLU function. It has previously been shown that DNN verification is NP-complete, even for simple specifications [28, 45]. However, when certain restricted classes of DNN architectures and specifications are considered, DNNs with ReLUs only can be verified in polynomial time [15]. The verification complexity and computability in the case of reactive systems controlled by ReLU DNNs (i.e., the DNN acts as an agent that repeatedly interacts with an environment) has also been studied recently [2, 3]. One recent work showed that verifying CTL properties in this context is undecidable [1]. In our work, however, we consider DNNs as stand-alone functions.



(a) An atom predicate of the form $\top$.

(b) An atom predicate of the form $P := \sum_i c_i \cdot y_i + b \geq 0$.

(c) A formula of the form $\varphi = \psi \wedge \theta$.

(d) A formula of the form $\varphi = \neg\psi$.

Figure 3 Constructs for each step of the induction.

When considering realistic implementations of the ReLU function, e.g., in *quantized neural networks* [24], the DNN verification problem for bit-vector specifications is PSPACE-hard, introducing a big complexity gap from the case of ideal mathematical form. When specific models of *graph neural networks* [62] are considered, the verification problem is undecidable [46].

For DNNs with Sigmoidal activation functions, two main results are presently known. First, it was shown that reachability analysis with some error tolerance ϵ, for any Lipschitz-continuous activation function, is NP-complete in the size of the network and of ϵ [44]. Second, it was shown that the decidability of $\mathcal{T}_{\text{exp}}$ implies the decidability of verifying DNNs with Sigmoidal activation functions, and that verifying such DNNs with a single hidden layer is decidable [27]. Our work here is another step towards a better understanding of the complexity of verifying such DNNs. The computational power of *Recurrent* Neural Networks with Sigmoidal activation functions has been studied as well, with Turing completeness results for Sigmoidal RNNs [9]. This can be further used to study the verification of Sigmoidal RNNs.

The complexity of formal analysis of DNNs with other functions, such as *Gaussian* and *arctan* has also been studied, showing the verification problem is at least as hard as deciding formulas in $\mathcal{T}_{\mathbb{R}}$ [60].

The connections between DNN verification and DNN reachability have also been studied before. Most prominently, it was shown that any local-robustness verification query can be reduced to a DNN reachability query [44]. In addition, a similar construction showing the equivalence between verification and reachability has been used before [13], though for a specific example without a formal proof.

6 Conclusion and Future Work

Our results show that for DNNs with ReLU, σ, tanh and NLReLU activation functions, the decidability of the verification problem is equivalent to a well-known open problem; and that it can be reduced to a problem whose approximation is decidable, and for whose complexity an upper bound is known. This was achieved by reducing the verification problem to the corresponding reachability problem. These results show a significant difference between the verification problem for DNNs with piecewise-smooth activation functions and for DNNs with piecewise-linear activation functions, which is known to be NP-complete [28, 45].

Moving forward, one goal that we plan to pursue is a version of our first result that does not rely on the NLReLU function, which is not as mainstream as the other functions that we considered. Although discarding this function does not alter the first direction of the proof, the second reduction currently requires it; and we plan to circumvent this requirement by defining a reduction from a Σ_{exp}-formula to a formula in the signature of $\mathcal{T}_{\mathbb{R}} \cup Th(\mathbb{R}, +, -, \cdot_{q\in\mathbb{Q}}, 0, 1, <, \sigma, \tanh)$, and then using a combination of the decision procedures for these two theories. It is noteworthy that the Nelson-Oppen method [38] cannot be directly applied here, since it requires the combined theories to be disjoint, which is not the case. Several generalizations of the Nelson-Oppen method for non-disjoint theories have previously been proposed [19, 39, 43, 53], and we speculate that these could be useful in this context.

Another interesting direction that we plan to pursue is to combine our work with approaches for switching between different kinds of machine learning models. For example, it would be intriguing to study whether DNNs with smooth activation functions can be reduced to decision trees or to neural networks with a fixed number of layers, as can apparently be done for piecewise-linear DNNs [5, 56]. Equivalently, fundamental differences between piecewise-linear DNNs and smooth DNNs might imply similar differences between other classes of machine learning models.

Our second result could also be generalized, in two different manners. First, our construction could be applied to verification queries that involve multiple DNNs, e.g., verification queries used for proving DNN equivalence [37]. This is true since for two DNNs $\mathcal{N}_1, \mathcal{N}_2$ operating on the same domain, the verification query $\mathcal{N}_1(x) \overset{?}{=} \mathcal{N}_2(x)$ can be reduced to $\mathcal{N}'(x) \overset{?}{=} 0$, where $\mathcal{N}'$ is constructed from copies of $\mathcal{N}_1, \mathcal{N}_2$ with outputs y_1, y_2, and where additional neurons are used to stipulate that $y_3 \geq 0 \iff y_1 = y_2$, using our construction. In this case, we have that $\mathcal{N}' = O(|\mathcal{N}_1| + |\mathcal{N}_2|)$. An illustration of this construction appears in Figure 4. These results, in turn, could be generalized to queries that involve any finite number of DNNs.

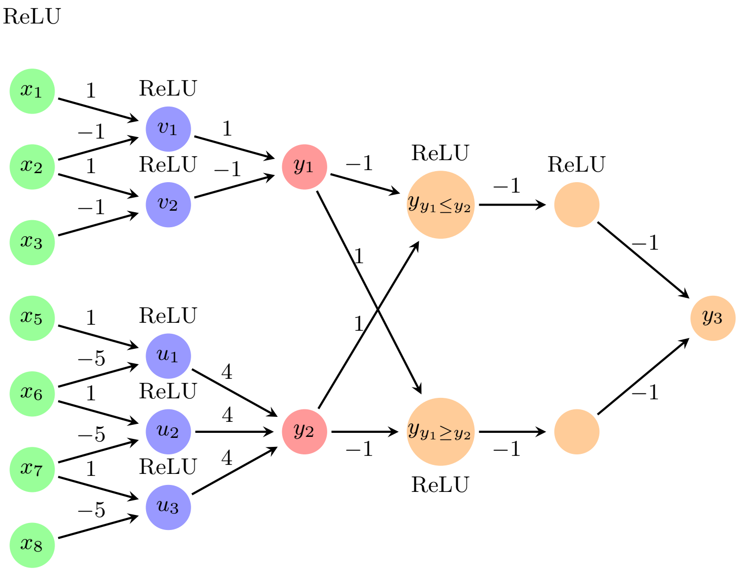


Figure 4 Reducing DNN equivalence to DNN reachability.

Second, similar constructions can support specification formulas over arithmetics that include any activation function (even piecewise-smooth). In this case, the number of added neurons is proportional not only to the sizes of the original network and the formula, but also to the number of activation functions composing the atoms. This construction is straightforward, and is omitted.

Our second result provides a notion of estimation for the DNN verification problem in general. That is, we could relax any DNN verification query to its equivalent ϵ-tolerant reachability query. This effectively allows an error tolerance in the value of the output neuron of the resulting network. However, the intuitive definition for approximating DNN verification is to introduce error tolerance to the values of all neurons. To that end, we plan to investigate the connections between these two definitions of relaxation, and the advantages of using each one.

A final line of work that we intend to pursue in the future is to consider a more realistic framework of verification, with a concrete implementation of σ, rather than its pure mathematical form. This is similar to what was done for DNNs with ReLU activation functions [24]. In addition, we intend to characterize *decidable fragments* of the DNN verification problem, by restricting specifications and/or architectures; that is, we plan to identify sufficient conditions on the DNNs and specifications, which would render the resulting verification problem decidable. For such decidable fragments, studying the computational complexity of the verification problem is yet another intriguing line of work. Similar research was conducted in the context of differential privacy [7], and it is interesting to study whether the decidable fragments identified in this research could be useful for DNN verification as well. We also intend to further explore implications of the *Quasi-Decidability* of $\mathcal{T}_{exp}$ [16] on DNN verification.

References

1 M. Akintunde, E. Botoeva, P. Kouvaros, and A. Lomuscio. Formal Verification of Neural Agents in Non-Deterministic Environments. *Autonomous Agents and Multi-Agent Systems*, 36:1–36, 2022.

2 M. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 197–210, 2019.

3 M. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano. Reachability Analysis for Neural Agent-Environment Systems. In *Proc. 16th Int. Conf. on Principles of Knowledge Representation and Reasoning*, 2018.

4 G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.

5 Caglar Aytekin. Neural Networks are Decision Trees, 2022. Technical Report. `arXiv:2210.05189`.

6 T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks And its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1249–1264, 2019.

7 Gilles Barthe, Rohit Chadha, Paul Krogmeier, A Prasad Sistla, and Mahesh Viswanathan. Deciding Accuracy of Differential Privacy Schemes. In *Proc. ACM on Programming Languages (POPL)*, pages 1–30, 2021.

8 M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. `arXiv:1604.07316`.

9 Jérémie Cabessa. Turing Complete Neural Computation Based on Synaptic Plasticity. *PloS one*, 14(10), 2019.

10 OpenAI. ChatGPT: Optimizing Language Models for Dialogue, 2022. URL: `https://openai.com/blog/chatgpt`.

11 Mingzhe Chen, Ursula Challita, Walid Saad, Changchuan Yin, and Mérouane Debbah. Artificial Neural Networks-Based Machine Learning for Wireless Networks: A Tutorial. *IEEE Communications Surveys & Tutorials*, 21(4):3039–3071, 2019.

12 S. Dubey, S. Singh, and B. Chaudhuri. Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. *Neurocomputing*, 2022.

13 Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Intl. Conf. Computer Aided Verification (CAV)*, pages 43–65, 2020.

14 A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. A Guide to Deep Learning in Healthcare. *Nature Medicine*, 25(1):24–29, 2019.

15 James Ferlez and Yasser Shoukry. Bounding the Complexity of Formally Verifying Neural Networks: a Geometric Approach. In *Proc. 60th IEEE Conf. on Decision and Control (CDC)*, pages 5104–5109, 2021.

16 Peter Franek, Stefan Ratschan, and Piotr Zgliczynski. Satisfiability of Systems of Equations of Real Analytic Functions is Quasi-Decidable. In *Proc. 36th Int. Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 315–326, 2011.

17 D. Fremont, J. Chiu, D. Margineantu, D. Osipychev, and S. Seshia. Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 122–134, 2020.

18 T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

19 S. Ghilardi. Model-Theoretic Methods in Combined Constraint Satisfiability. *Journal of Automated Reasoning*, 33:221–249, 2004.

20 I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, 2016.

21 I. Goodfellow, J. Shlens, and C Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. `arXiv:1412.6572`.

22 Dario Guidotti, Luca Pulina, and Armando Tacchella. pyNeVer: A Framework for Learning and Verification of Neural Networks. In *Proc. 19th Int. Symposium Automated Technology for Verification and Analysis (ATVA)*, pages 357–363, 2021.

23 P. Henriksen and A. Lomuscio. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 2513–2520, 2020.

24 T. Henzinger, M. Lechner, and Đ. Žikelić. Scalable Verification of Quantized Neural Networks. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 3787–3795, 2021.

25 X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.

26 Omri Isac, Yoni Zohar, Clark Barrett, and Guy Katz. DNN Verification, Reachability, and the Exponential Function Problem, 2023. Technical Report. `arXiv:2305.06064`.

27 R. Ivanov, J. Weimer, R. Alur, G. Pappas, and I. Lee. Verisig: Verifying Safety Properties of Hybrid Systems with Neural Network Controllers. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, pages 169–178, 2019.

28 G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.

29 X. Liu, L. Xie, Y. Wang, J. Zou, J. Xiong, Z. Ying, and A. Vasilakos. Privacy and Security Issues in Deep Learning: A Survey. *IEEE Access*, 9:4566–4593, 2020.

30 Yang Liu, Jianpeng Zhang, Chao Gao, Jinghua Qu, and Lixin Ji. Natural-Logarithm-Rectified Activation Function in Convolutional Neural Networks. In *Proc. 5th Int. Conf. on Computer and Communications (ICCC)*, pages 2000–2008, 2019.

31 Alessio Lomuscio and Lalit Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. `arXiv:1706.07351`.

32 A. Lukina, C. Schilling, and T. Henzinger. Into the Unknown: Active Monitoring of Neural Networks. In *Proc. 21st Int. Conf. Runtime Verification (RV)*, pages 42–61, 2021.

33 Z. Lyu, C.-Y. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 5037–5044, 2020.

34 A. Macintyre and A. Wilkie. On the Decidability of the Real Exponential Field. *Kreisel's Mathematics*, 115:451, 1996.

35 M. Müller, C. Brix, S. Bak, C. Liu, and T. Johnson. The Third International Verification of Neural Networks Competition (VNN-COMP 2022): Summary and Results, 2022. Technical Report. `arXiv:2212.10376`.

36 M. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In *Proc. 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–33, 2022.

37 N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh. Verifying Properties of Binarized Deep Neural Networks, 2017. Technical Report. `arXiv:1709.06662`.

38 G. Nelson and D. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

39 Enrica Nicolini, Christophe Ringeissen, and Michaël Rusinowitch. Data Structures with Arithmetic Constraints: A Non-Disjoint Combination. In *Proc. 7th Int. Symposium of Frontiers of Combining Systems (FroCoS)*, pages 319–334, 2009.

40 M. Ostrovsky, C. Barrett, and G. Katz. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In *Proc. 20th. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 391–396, 2022.

41 L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.

42 D. Richardson. Some Undecidable Problems Involving Elementary Functions of a Real Variable. *The Journal of Symbolic Logic*, 33(4):514–520, 1969.

43 Christophe Ringeissen. Cooperation of Decision Procedures for the Satisfiability Problem. In *Proc. 1st Int. Workshop of Frontiers of Combining Systems (FroCoS)*, pages 121–139, 1996.

44 Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees, 2018. Technical Report. `arXiv:1805.02242`.

45 Marco Sälzer and Martin Lange. Reachability Is NP-Complete Even for the Simplest Neural Networks. In *Proc. 15th Int. Conf. on Reachability Problems (RP)*, pages 149–164, 2021.

46 Marco Sälzer and Martin Lange. Fundamental Limits in Formal Verification of Message-Passing Neural Networks. In *Proc. 11th Int. Conf. on Learning Representations (ICLR)*, 2023.

47 S. Sankaranarayanan, S. Dutta, and S. Mover. Reaching Out towards Fully Verified Autonomous Systems. In *Proc. 13th Int. Conf. on Reachability Problems (RP)*, pages 22–32, 2019.

48 J. Shoenfield. *Mathematical Logic.* Addison-Wesley publishing, 1967.

49 G. Singh, T. Gehr, M. Püschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–30, 2019.

50 C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. `arXiv:1312.6199`.

51 Alfred Tarski. *Project Rand: A Decision Method for Elementary Algebra and Geometry.* Rand Corporation, 1948.

52 Alfred Tarski. A Decision Method for Elementary Algebra and Geometry. *Journal of Symbolic Logic*, 17(3):24–84, 1952.

53 Cesare Tinelli and Christophe Ringeissen. Unions of Non-Disjoint Theories and Combinations of Satisfiability Procedures. *Theoretical Computer Science*, 290(1):291–353, 2003.

54 H.-D. Tran, S. Bak, W. Xiang, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 18–42, 2020.

55 A. Turing. Computing Machinery and Intelligence. *Mind*, LIX(236), 1950.

56 Mattia Villani and Nandi Schoots. Any Deep ReLU Network is Shallow, 2023. Technical Report. `arXiv:2306.11827`.

57 S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.

58 Alex Wilkie. Model Completeness Results for Expansions of the Ordered Field of Real Numbers by Restricted Pfaffian Functions and the Exponential Function. *Journal of the American Mathematical Society*, 9(4):1051–1094, 1996.

59 H. Wu, A. Zeljić, G. Katz, and C. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 143–163, 2022.

60 Adrian Wurm. Complexity of Reachability Problems in Neural Networks, 2023. Technical Report. `arXiv:2306.05818`.

61 H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.

62 Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph Neural Networks: A Review of Methods and Applications. *AI open*, 1:57–81, 2020.

Faithful Simulation of Randomized BFT Protocols on Block DAGs

Hagit Attiya
Technion, Haifa, Israel

Constantin Enea
LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, France

Shafik Nassar
Technion, Haifa, Israel

Abstract

Byzantine Fault-Tolerant (BFT) protocols that are based on *Directed Acyclic Graphs* (DAGs) are attractive due to their many advantages in asynchronous blockchain systems. These DAG-based protocols can be viewed as a *simulation* of some BFT protocol on a DAG. Many DAG-based BFT protocols rely on randomization, since they are used for agreement and ordering of transactions, which cannot be achieved deterministically in asynchronous systems. Randomization is achieved either through local sources of randomness, or by employing shared objects that provide a common source of randomness, e.g., *common coins*.

A DAG simulation of a randomized protocol should be *faithful*, in the sense that it precisely preserves the properties of the original BFT protocol, and in particular, their probability distributions. We argue that faithfulness is ensured by a *forward simulation*. We show how to faithfully simulate any BFT protocol that uses public coins and shared objects, like common coins.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; General and reference → Verification

Keywords and phrases Byzantine failures, Hyperproperties, Forward Simulation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.27

Related Version *Extended Version*: `https://eprint.iacr.org/2023/192`

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).
Constantin Enea: partially supported by the project AdeCoDS of the French ANR Agency.
Shafik Nassar: partially funded by the European Union (ERC, FASTPROOF, 101041208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Asynchronous distributed computation is naturally captured by a *directed acyclic graph* (DAG), whose nodes describe local computation and edges correspond to causal dependency between computation at different processes. Lamport's *happens-before* relation [14] is an example of such DAG, where each node is a single local computation event, and each edge is a single message delivery event. *Block* DAGs [21] go one step further and incorporate more than one local computation step in each block (node); these steps may even belong to several *independent* protocols.

By exchanging blocks in a manner that preserves their dependencies, a distributed protocol can now be abstracted as a joint computation of a block DAG. In particular, a general *Byzantine fault-tolerant* (BFT) DAG-based algorithm combines two components:

one component builds the DAG using a communication protocol that tolerates malicious failures, and the other component performs the local computation embodied in each node of the DAG. The first component can be used to separate the task of injecting user input to the system, such as transactions, from the task of processing these inputs and producing an output, e.g., an ordering of those transactions.

This generality makes block DAGs an attractive approach for designing coordination protocols for, e.g., Byzantine Atomic Broadcast [10, 13, 20], consensus [4, 16] and cryptocurrencies [6]. (For a survey of the techniques used in block DAG approaches, see [21].) A block DAG can be seen as a strict extension of a *blockchain*, which is a DAG where all blocks are *totally ordered*, i.e., a directed path. The DAG approach was shown to achieve high throughput [19] due to the flexibility it provides over the standard blockchain approach.

Schett and Danezis [17] show that any *deterministic* BFT protocol can be simulated as a block DAG. They provide generic mechanisms for processes to maintain a consistent view of the block DAG, and to individually *interpret* the DAG as an execution of some protocol.

The restriction to deterministic protocols, however, handicaps the applicability of this result, since many algorithms in the asynchronous domain are necessarily non-deterministic, due to the FLP impossibility result [9]. For example, DAG-based agreement protocols with provable security, like Aleph [10] or DAG-Rider [13], are either randomized or assume the existence of a shared source of randomness. This calls for a framework that can handle *randomized* BFT protocols; those that either utilize local randomness or even a shared object.

The problem of using or defining block DAG simulations in the context of *randomized protocols* has two aspects: (1) using a block DAG simulation of a *deterministic* protocol as a building block of a *randomized protocol*, and (2) defining block DAG simulations of *randomized protocols.*

Concerning the first aspect above, we aim to enable modular reasoning when using such simulations instead of the original protocols (Section 2 describes a concrete example). Schett and Danezis [17] establish that the traces of the block DAG simulation are included in the set of traces of the original protocol (for some notion of trace which is not important for this discussion). However, as shown in other contexts, e.g., concurrent objects [2, 11], such a notion of refinement is not sufficient to conclude that relevant specifications of a randomized protocol that builds on some other deterministic protocol are preserved when the latter is replaced by the block DAG simulation. Indeed, the specifications of randomized protocols characterize sets (probabilistic distributions) of executions and are instances of *hyper-properties* which are not preserved by standard trace inclusion [2].

Therefore, we establish a stronger notion of refinement between a block DAG simulation and the original protocol, namely, that there exists a *forward simulation* between the two. (A forward simulation maps every step of one protocol to a sequence of steps of the other protocol, starting from the initial state of the first and advancing in a forward manner; a backward simulation is similar, but it goes in the reverse direction, from end states back to initial states.). Based on the results in [2], this implies that any finite-trace specification of a randomized protocol against an adaptive adversary is preserved when a sub-protocol is replaced by its block DAG simulation. We recall that an *adaptive adversary* is a scheduler that resolves all the non-determinism introduced by the interleaving semantics and which can observe everything about the local state of a process or the messages in transit.

Armed with this understanding of the precise nature of block DAG simulation, we present an extension of the construction of Schett and Danezis [17], which applies also to protocols using randomization and shared objects. Specifically, we consider *randomized* protocols in which the local coin flips of each process may be public, we call those protocols *public-coin* protocols. We prove that any public-coin protocol that uses shared objects, e.g., common coins, can be simulated on a block DAG, preserving its usage of shared objects.

Algorithm 1 Binary consensus using a common coin.

```
Input: $x$
$r := 0$; $est := x$;
while $true$ do
    $r{+}{+}$;
    $val := r.\mathsf{BCA}(est)$;
    $c := r.\mathsf{Toss}()$;
    if $val \neq \bot$ and $c = val$ then
        output $val$;
        $est := val$;
    else if $val \neq \bot$ then
        $est := val$;
    else
        $est := c$;
```

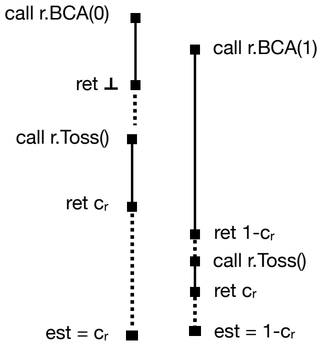


Figure 1 A randomized consensus algorithm on the left, and an execution template ($c_0 \in \{0, 1\}$) on the right, which represents the executions of an adaptive adversary which disallows termination.

A relationship based on a forward simulation allows to conclude that probabilistic specifications of a randomized protocol, e.g., termination time, are preserved by its block DAG simulation. Such a simulation precisely preserves the finite trace distribution and the probabilistic relationship between inputs and outputs. This means that whatever "adverse" effects can occur in the simulation, can already be demonstrated in the original protocol.

Organization. Section 2 presents an example that demonstrates why simulations should preserve hyperproperties. Sections 3 and 4 describe the model and introduce important definitions and notations. Section 5 formally defines block DAGs. Our results are presented and proved in Section 6. The relation of our simulation to the work of Schett and Danezis [17], and some applications appear in Section 7. We summarize with future work, in Section 8.

2 Motivating Example

We describe a class of protocols solving *Binary Crusader Agreement*, and a hyperproperty about them, called *binding* [1], which is assumed when such protocols are used to solve randomized consensus. This motivates the need for establishing a notion of refinement for block DAG simulations that is stronger than trace inclusion and which enables the preservation of such hyperproperties.

Randomized consensus based on Binary Crusader Agreement. Let us consider the consensus protocol listed in Algorithm 1 (from [1]). This is a randomized protocol based on two sub-protocols, Binary Crusader Agreement, invoked as BCA, and a common coin, invoked via Toss. Every process participating in this consensus protocol goes through a sequence of asynchronous rounds (the current round is stored in the variable r), and each round consists of one instance of BCA followed by one instance of Toss. We prefix invocations with the value of r in order to emphasize that these instances are different from one round to another.

Binary Crusader Agreement [7] is a weak form of consensus, where processes start with a value in $\{0, 1\}$ and can return a value in $\{0, 1, \perp\}$ (note the special value $\perp$). The requirements are: (1) validity: if all non-faulty processes start with the same input, then this is the only output, (2) agreement: no two non-faulty processes output two distinct non-$\perp$ values, and (3) termination: every non-faulty process eventually outputs a value. It is weaker than consensus because a process can output the "don't know" value $\perp$ instead of one of the inputs. The common coin protocol allows to implement a shared source of *uniform* randomness, it guarantees that all processes receive the same output in $\{0, 1\}$ (drawn with equal probability) and that this output is unpredictable to an outsider (adversary).

Each round of the consensus protocol starts with a round of BCA where each process inputs the current estimation of the agreement value *est* (initially, this is the input x), followed by a round of the common coin. If BCA returns a non-$\perp$ value then this will be the value of *est* in the next round. Otherwise, the value of *est* is the value returned by the coin protocol. Furthermore, if the values returned by BCA and Toss are the same, then the process outputs the decision value. A process continues running the protocol after outputting the decision in order to "help" other processes reach a decision (e.g., so that future instances of BCA and the common coin satisfy honest super majority assumptions).

Termination under binding. We say that the protocol *terminates* when all non-faulty processes output a decision. It has been shown [1] that the protocol of Figure 1 terminates against an adaptive adversary with probability 1, provided that BCA satisfies a property called *binding.* The binding property states that for every execution prefix of BCA that ends with a process returning $\perp$, there is a *single* non-$\perp$ value that can be returned by a process in *any* future extension of this prefix. It is important to note that this is an instance of a *hyperproperty* because it characterizes *sets* of executions, i.e., all possible extensions of a prefix, instead of individual executions as in standard safety or liveness properties.

To explain the usefulness of binding, we use the execution template on the right of Figure 1. This defines non-terminating executions of the consensus protocol against a specific adaptive adversary assuming a "worst-case" BCA protocol, which satisfies the specification described earlier but does *not* satisfy binding. Therefore, assuming two processes with different inputs, for every round r, the adversary schedules BCA so that a first process returns $\perp$ and the second process's return value is not yet fixed. Then, it schedules the first process to get a value $c_r \in \{0, 1\}$ from the common coin and after observing this value, it resumes BCA so that the second process gets the value $1 - c_r$ (this is admitted by the BCA specification). The conditional at lines 6–12 implies that the first process will enter the next round with *est* being the outcome of the coin toss, and the second process with *est* being the value returned by BCA. Therefore, they enter the next round with different estimations of the agreement value, and the same can be repeated infinitely often. Since this repeats for all possible outcomes of the coin tosses, non-termination happens with probability 1.

Note that this would not be possible for both outcomes $c_r \in \{0, 1\}$ of the coin toss if BCA satisfies binding. Indeed, after the first process gets $\perp$ from BCA (and before the coin toss), the value returned by BCA to the second process is *fixed* in *any* possible extension, i.e., it is the same no matter the outcome of the coin toss. Therefore, for one of the two possible outcomes of the coin toss, this return value equals that outcome, and the two processes will enter with equal values of *est* in the next round.

When binding holds, an adaptive adversary can *not* impose the schedule described above and the protocol terminates with probability 1. In every round, if the BCA value is not $\perp$, then it equals the outcome of the coin toss with probability 1/2, which leads to outputting a decision. If all processes get $\perp$ from BCA, then the common coin leads directly to agreement. Therefore, the protocol terminates within a constant expected number of rounds.

Preserving binding. In the context of this consensus protocol, we discuss the possibility of replacing a given BCA protocol with a block DAG simulation as defined by Schett and Danezis [17]. The results in [17] are not sufficient to deduce that the block DAG simulation satisfies binding if the original protocol did, because, as mentioned above, binding is an instance of a hyper-property and hyper-properties are not preserved by standard trace inclusion [2]. Therefore, based on the results in [17], the proof of termination that assumed binding is not applicable to the block DAG simulation.

In this work, we present a block DAG simulation that handles protocols that use public-coins and shared objects (including a common coin like Toss). We establish that it is a *forward simulation*, which by previous work [2], implies that the set of traces defined by an adaptive adversary of the consensus protocol with the original BCA protocol is the same when the latter is replaced with the block DAG simulation (the results in [2] were applied in the context of concurrent objects and programs using such objects, but they are stated in terms of LTSs models of such programs and apply more generally to distributed protocols as well). Therefore, if one satisfies binding, then the other one satisfies it as well. This is enough to conclude that the termination argument used for the original protocol holds for the block DAG simulation as well.

3 Preliminaries

For any $n \in \mathbb{N}$, we denote $[n] = \{1, \ldots, n\}$. For any two strings s_1 and s_2, we denote by $s_1 \circ s_2$ the concatenation of the two strings.

We consider an asynchronous network with n processes $p_1, \ldots, p_n$. Each process p_i has a local process state PS_i, and buffers $In_{j \to i}$ and $Out_{i \to j}$, for each $j \in [n]$, that serve for communicating with p_j, as well as a buffer $Rqsts_i$ that contains incoming user requests. A schedule consists of two types of events:

- A $\mathsf{compute}(i)$ event lets process p_i receive *all* the messages in the buffers $In_{j \to i}$, as well as the requests in $Rqsts_i$, and update the local state PS_i. The local computation performed to update PS_i may result in new messages being deposited in the outgoing buffers $Out_{i \to j}$ and indications being sent to the user.
- A $\mathsf{deliver}(i, j)$ event moves the *oldest* message in $Out_{i \to j}$ to $In_{j \to i}$.

We assume a computationally bounded adversary that may adaptively corrupt up to f processes, and also controls the scheduling of the system. Initially, all n processes are *correct* and honestly follow the protocol. Once a process is corrupted, it may behave arbitrarily. The adversary can also read all messages in the system, even those sent by correct processes. Although the scheduling of message delivery is adversarial, we assume eventual delivery, i.e., every message sent is eventually delivered.

In a randomized protocol, the local computation of a process can depend on the result of local coin flips. To model this, we assume each process p_i has access to a random *tape*, from which it can draw a random string at each $\mathsf{compute}(i)$ event. Our simulation can be applied to *public-coin* protocols, which are randomized protocols that do no require processes to keep secrets, i.e., they can broadcast the random string they draw as soon as they use it. This definition captures protocols in the full-information model such as [12].

To allow for easy composition, we define *shared objects*. A shared object is an implementation of an interface that is accessible by all processes. For example, in the context of the randomized consensus protocol in Fig. 1 we used a shared object called *common coin* with a method Toss. For any shared object o, each process p_i can invoke o as it performs any local computation. Invocations are non-blocking, and o may at any point return a value in a designated buffer $\mathsf{o}.\mathit{buff}_i$. Whenever a $\mathsf{compute}(i)$ event is scheduled, the contents of $\mathsf{o}.\mathit{buff}_i$ are dequeued and may affect the local computation.

4 Modeling protocols with Labeled Transition Systems

We model a protocol as a *Labeled Transition System* (*LTS*), which is a tuple $L = (Q, \Sigma, q_{start}, \delta)$ where Q is a (possibly infinite) set of states, Σ is a set of (transition) labels, q_{start} is the starting state, and $\delta \subseteq Q \times \Sigma \times Q$ is a (possibly infinite) set of transitions, written as $q_1 \xrightarrow{l} q_2$ for any $(q_1, l, q_2) \in Q \times \Sigma \times Q$.

An execution of L is an alternating sequence of states and transition labels $\alpha = q_0, l_0, q_1, l_1, \ldots$ s.t. $q_i \xrightarrow{l_i} q_{i+1}$ for any $i \geq 0$. If there is a partial execution $q_i, l_i, \ldots, l_{j-1}, q_j$ then we write $q_i \xrightarrow{l_i,\ldots,l_{j-1}} q_j$. We define a subset of labels $\Sigma_E \subseteq \Sigma$ as the *external actions*, and define a *trace* of L to be the projection of an execution over Σ_E. Typically, external actions correspond to requests and indications in the interface of a protocol, and define the "observable" behavior of a protocol. For instance, the external actions of a consensus protocol are about setting the input of each process and outputting their decisions.

LTSs can easily be used to model deterministic protocols. Essentially, LTS states correspond to tuples of states of participating processes and communication channels, and each transition corresponds to a step of some process (more details are given below).

Randomized protocols can be modeled using an extension of LTSs called *(simple) probabilistic automata* [18] where a transition from a state q leads to a probability distribution over states instead of a single state. The semantics of a probabilistic automaton is formalized in terms of *probabilistic executions*, which are probability distributions over executions defined by a deterministic scheduler that resolves the non-determinism. *Probabilistic traces* are defined as projections of probabilistic executions to external actions (similarly to the non-probabilistic case). The deterministic scheduler corresponds to the notion of adaptive adversary described above which controls message delivery and process scheduling. To simplify the formalization, we model randomized protocols using LTSs instead of probabilistic automata by including results of random choices in the transition labels. The transition labels corresponding to random choices are defined as external actions. The relevance of this modeling choice will be detailed later when discussing forward simulations.

Let $\mathcal{P}$ be a public-coin protocol and $\mathcal{O}$ be a *set* of shared objects used by $\mathcal{P}$. We define the LTS of $\mathcal{P}$ as follows $L = (Q, \Sigma, q_{start}, \delta)$. A state $q \in Q$ consists of the local state PS_i, the incoming messages $(In_{j\to i})_{j\in[n]}$, the outgoing messages $(Out_{i\to j})_{j\in[n]}$ and the incoming object return values $(\mathsf{o}.\mathit{buff}_i)_{\mathsf{o}\in\mathcal{O}}$ of each process p_i. For convenience, we assume that incoming user requests are stored in $In_{i\to i}$ and outgoing user indications are stored in $Out_{i\to i}$. Overall, $q = \big(PS_i, (In_{j\to i})_{j\in[n]}, (Out_{i\to j})_{j\in[n]}, (\mathsf{o}.\mathit{buff}_i)_{\mathsf{o}\in\mathcal{O}}\big)_{i\in[n]}$. We use register notation to refer to the components of each state, e.g., $q.In_{j\to i}$ refers to the incoming messages buffer from j to i in the state q. In the initial state q_{start}, all of the processes have the initial local state and all of the message buffers are empty. For the consensus protocol in Fig. 1, local states are valuations of r, val, c, and est, and the buffer for incoming object return values will contain values returned by Toss. User indications are decision values outputted at line 7.

The transition labels Σ correspond to the different types of steps in a protocol execution, namely, local computation, message delivery, return values from objects in $\mathcal{O}$, or user requests and indications. Observe that we do not need to label sending requests to $\mathsf{o} \in \mathcal{O}$ as this is done in an ordinary local computation event. In addition, the local computation label would include the randomness (if any) that is used by the process in the said computation event.

Formally, the labels in Σ are as follows:

1. $\mathsf{compute}(i, \rho)$ denotes a transition where process p_i performs a local computation with ρ as its randomness. For the consensus protocol in Fig. 1, a local computation step would consist in assigning a value to *est* depending on the conditions starting with line 6.
2. $\mathsf{deliver}(i \to j)$ denotes a transition where all messages in $Out_{i\to j}$ are moved to $In_{i\to j}$.
3. $\mathsf{o.indicate}(i, w)$ denotes a transition where the value w has been added to $\mathsf{o}.\mathit{buff}_i$. In Fig. 1, this would correspond to the common coin object returning a value for Toss.
4. $\mathsf{request}(i, x)$ denotes a transition where process p_i receives x as input. In Fig. 1, this models a process receiving an input value to use in the consensus protocol.
5. $\mathsf{indicate}(i, y)$ denotes a transition where process p_i returns y as output. In Fig. 1, this corresponds to the output at line 7.

The external actions in $\Sigma_E \subseteq \Sigma$ are user requests ($\mathsf{request}(i, x)$) and indications ($\mathsf{indicate}(i, y)$), and local computation events ($\mathsf{compute}(i, \rho)$). The latter are included in Σ_E in order to be able to relate probability distributions in different protocols, as discussed hereafter. A transition $(q_1, l, q_2) \in Q \times \Sigma \times Q$ is in δ if and only if the protocol can get from state q_1 to state q_2 by executing the step denoted by the label l.

Showing that a block DAG protocol is a "correct" simulation of some other protocol relies on the notion of *forward simulation* between the LTSs modeling the two protocols.

▶ **Definition 1** (forward simulation). *Let $L = (Q, \Sigma, q_{start}, \delta)$ and $L' = (Q', \Sigma', q'_{start}, \delta')$ be two LTSs with the same set of external actions Σ_E. A relation $R \subseteq Q \times Q'$ is a* forward simulation *from L to L' if both of the following hold:*

- $(q_{start}, q'_{start}) \in R$
- *For any $(q_1, l, q_2) \in \delta$ and any q'_1 such that $(q_1, q'_1) \in R$, there exists $q'_2 \in Q'$ such that:*
  - $(q_2, q'_2) \in R$,
  - *$q'_1 \xrightarrow{\sigma} q'_2$ is a partial execution of L' (σ is a sequence of labels in Σ'), and*
  - *if $l \in \Sigma_E$, then the projection of the label sequence σ over Σ_E is exactly l.*

When L is an LTS modeling a block DAG simulation of a deterministic protocol $\mathcal{P}$ that is modeled as an LTS L', the existence of a forward simulation R from L to L' implies that the set of traces of L is included in the set of traces of L' [15]. It also implies the preservation of (hyper-)properties of *finite* probabilistic traces of randomized protocols when some sub-protocol $\mathcal{P}$ is replaced by a block DAG simulation of it [2] (a concrete example was given in Section 2). If the forward simulation is *weak progressive* [8], i.e., there exists a well-founded order such that if $\sigma = \epsilon$ in Definition 1 then either q_2 is smaller than q_1 in this order or there exists an infinite execution from q'_2 with empty trace, then (hyper-)properties of *infinite* probabilistic traces are also preserved.

These results extend to randomized protocols as well. Assuming that the random choices follow the uniform distribution, a forward simulation would imply that any random choice in L is mimicked in precisely the same manner by L'. This is because the label of every step that includes a random choice is an external action and the result of that random choice is included in the label itself. This holds even for *non-uniform* random sampling as long as probabilities are recorded in transition labels. More formally, it will imply the existence of a *weak probabilistic simulation* which is known to imply that the probability distributions over traces of L defined by a deterministic scheduler are included in the probability distributions over traces of L' defined by a deterministic scheduler [18]. Moreover, it will also imply the preservation of probability distributions over executions of programs that use the block DAG simulation instead of the original protocol (this is a consequence of weak probabilistic simulations being sound for the trace distribution precongruence [18]).

It follows that any standard specification of a protocol, e.g., safety or (almost-sure) termination against an adaptive adversary, is preserved by a block DAG simulation provided the existence of a forward simulation. Moreover, typical specifications of programs using the DAG simulation instead of the original protocol will also be preserved.

5 Block DAGs

A *block* is the main type of message that is exchanged in DAG-based protocols and our block DAG simulations. A block issued by some process p_i allows p_i to: (1) inject data into the system, e.g., user inputs or shared object outputs, and (2) establish a dependency between events of different processes. To that end, the main fields of a block B are the identity of the issuing process $B.p$, injected data $B.d$, and references to other blocks $B.preds$ (on which B directly depends). The reference of B is denoted by $\mathsf{ref}(B)$.

We require that each reference must uniquely identify a specific block. One way to achieve this is using *cryptographic collision resistant hash functions*: the reference $\mathsf{ref}(B)$ consists of a hash of the block B. By the collision resistance of the hash function, it is infeasible for a computationally bounded adversary (or correct processes) to issue two distinct blocks that hash to the same value and this ensures that the reference identifies a unique block.

Since blocks are supposed to represent local computation, and local computation steps of any one process are always totally ordered, then each block B must include one reference to a parent block which we denote by $B.parent$, except for one *genesis* block for each process which does not have a parent. In addition, all of the blocks issued by one honest process should form a chain, i.e., a directed path that starts with the genesis block.

We define the *ancestors* of a block B to be all of the predecessors of B, and their predecessors and so on; this set is denoted $\mathsf{ancestors}(B)$.

A block B is *authentic* if it was issued by the process $B.p$. It is crucial to ensure the authenticity of each block before allowing it into the system. Otherwise, faulty processes can impersonate honest processes and sabotage safety properties. We can ensure authenticity by using a *cryptographic digital signature scheme*. That is each process must sign each block it issues, and other processes validate the block by checking the signature attached to it.

Ensuring that each individual block is authentic is not enough to ensure that only authentic blocks enter the system. We should also require that a block depends only on authentic blocks, that is $\mathsf{ancestors}(B)$ must all be authentic in order for B to enter. We say that a block is *valid* if it is authentic and all of $B.preds$ are valid. Note that this recursive definition is equivalent to requiring $\mathsf{ancestors}(B)$ all be authentic. Following this discussion, to ensure safety, only valid blocks would be considered by correct processes. When a process p_i validates a block B, we write $\mathsf{valid}(p_i, B)$.

Each process p_i maintains a local DAG G_i consisting of the valid blocks that p_i receives as nodes and includes a directed edge $B' \to B$ if and only if $B' \in B.preds$. Note that we need a mechanism for p_i to ensure that G_i is a DAG. A simple mechanism would be for p_i to validate B only after it has validated $B.preds$ and not validate multiple blocks "atomically". This alongside the fact that each reference identifies a unique block, would ensure that no block in a directed cycle would ever be considered valid. Formally, a *Block DAG of a correct process* p_i is a graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ such that

- $V_\mathcal{G} \subseteq \{B : \mathsf{valid}(p_i, B)\}$.
- If $B \in V_\mathcal{G}$ then for all $B' \in B.preds$ it holds that $B' \in V_\mathcal{G}$.
- $E_\mathcal{G} = \{(B', B) \in V_\mathcal{G} \times V_\mathcal{G} : B' \in B.preds\}$.
- $\mathcal{G}$ is acyclic.

Observe that by the definition of $\mathcal{G}$, for every $B \in V_\mathcal{G}$ it holds that $\mathsf{ancestors}(B) \subseteq V_\mathcal{G}$. When $B' \in \mathsf{ancestors}(B)$, we write $\mathsf{path}(B', B)$.

6 Simulating Public-Coin Protocols That Use Shared Objects

Simulating a protocol on a block DAG consists of two components: first, a mechanism that allows processes to build and maintain a *joint block DAG* and second, an algorithm to *interpret* this joint block DAG as an execution of the original protocol. Given those two ingredients, we can execute an instance of the protocol without sending any actual messages that are specific to the protocol itself. Of course, maintaining the joint block DAG would require exchanging one type of message (block), but those messages are agnostic to the protocol being simulated. This means that we can use the same joint block DAG to interpret multiple instances of the same protocol or even instances of different protocols.

Figure 2 describes how to simulate a public-coin protocol $\mathcal{P}$ using the components mentioned above. We refer to this protocol as the *block DAG simulation of* $\mathcal{P}$ and denote it by $\mathsf{BD}(\mathcal{P})$. We allow $\mathsf{BD}(\mathcal{P})$ to access the same shared objects as $\mathcal{P}$.

Simulation of Public-Coin Protocols on Block DAGs

From the perspective of process p_i, user requests go directly to $Rqsts_i$.
Initially, $G_i = (\{B_j\}_{j\in[n]}, E_i)$, where B_j is a dummy genesis block for process p_j.
On every $\mathsf{compute}(i)$ event:

1. Run $\mathsf{genBlock}(G_i, blks)$.
2. If new blocks were added to G_i, then run $\mathsf{interpret}(G_i, \mathcal{P})$.
3. Run $\mathsf{exchangeBlocks}(G_i, blks)$.

Figure 2 The simulation algorithm for public-coin protocols.

Interpreting the block DAG as an execution of $\mathcal{P}$ is done using the interpret algorithm, described in Section 6. This algorithm runs locally and involves no communication, yet guarantees that if two correct processes are interpreting the same (partial) block DAG, then their interpretations would be identical.

Maintaining the joint block DAG is done using the genBlock and exchangeBlocks algorithms (discussed in Section 6): genBlock is responsible for creating new blocks and exchangeBlocks is responsible for passing those blocks around to ensure that all correct processes receive the same blocks even if the process that issued the block is corrupted.

The aforementioned components, together, ensure that correct processes have consistent views of the execution of $\mathcal{P}$ at all times. However, this does not guarantee that the execution is useful, e.g., it might give the adversary more power or it might be a "liveless" execution where the correct processes are not making any progress. For that reason, we prove in Section 6 that the execution (defined by the views) is faithful in the sense that there exists a forward simulation towards the original protocol. This guarantees that the simulation of $\mathcal{P}$ on the block DAG preserves $\mathcal{P}$'s original specification.

Common Interpretation. Given a block DAG $\mathcal{G} = (V, E)$, we want to interpret it as an execution of the protocol. We call this execution the *simulated execution.* Furthermore, we need the interpretation to be consistent among all correct processes doing it.

The idea is to view $\mathcal{G}$ as a causality graph, where a block in $\mathcal{G}$ issued by some process p_i corresponds to a node that belongs to p_i in the causality graph, and the node corresponds to a $\mathsf{compute}(i)$ in the simulated execution. In order to interpret $\mathcal{G}$, we interpret each block separately, where the interpretation of the block consists of the local process state and its outgoing messages after the corresponding $\mathsf{compute}(i)$ event. For convenience, we also treat the incoming messages (right before the event) as part of the interpretation. Formally:

▶ **Definition 2** (Block Interpretation). *The* interpretation *of a block B has the following fields:*

1. *A local process state $B.PS$.*
2. *A list of incoming messages $B.M_{in}$.*
3. *A list of outgoing messages $B.M_{out}$. For convenience, we denote by $M_{out}[j]$ the outgoing messages in M_{out} that are designated to p_j.*

Note that the interpretation of a block is *not* sent over the network. This is crucial because we do not want the size of the block sent over the network to increase with the number of protocol instances being interpreted, and instead we only want the block to include information that processes cannot locally compute unambiguously. As such, it is the responsibility of each process to interpret each block it has locally.

In a regular execution of a *deterministic* protocol, whenever a compute(i) event is scheduled, the process p_i performs the following: it passes all of the message in $In_{j\rightarrow i}$ to the local state of its protocol instance PS_i and performs a local computation. This updates the local state PS_i, produces new outgoing messages that are deposited into $Out_{i\rightarrow j}$ and may return user indications. Our interpretation protocol tries to mimic the execution by assigning to $B.PS$ the local state of the process after the corresponding event, $B.M_{out}[j]$ the messages that would be deposited in $Out_{i\rightarrow j}$, and $B.M_{in}$ the messages that would have been in $In_{j\rightarrow i}$ before the event. In addition, if the block B was issued by the process doing the interpretation and $B.PS$ produces a user indication, then the process must actually return the indication to the user. The way to compute $B.PS$ is as follows: $B.PS$ is initially copied from the parent block (or initialized as an initial state for genesis blocks), and then we feed it all of the relevant outgoing messages from the interpretation of the predecessor blocks, that is all messages in $B'.M_{out}[i]$ for all $B' \in B.preds$, where $B.p = p_i$.

When extending this approach to *randomized* protocols, we need to account for the local randomness. In this case, the process state expects to additionally receive a random tape. It is the responsibility of the issuing process to include the tape in the block B and attach it as a part of the block in a data field $B.rand$. The interpretation is thus similar to that of a deterministic protocol, but $B.rand$ is now also passed to the process state as randomness.

When further extending this to protocols with *shared objects*, we need to handle object invocations and object indications. In a regular execution of a protocol with a shared objects o, a process p_i might invoke o following a compute(i) event. Similarly, when interpreting a block, $B.PS$ might dictate that $B.p$ should invoke o. In this case, the interpreting process p_i actually performs the invocation only if it is the issuing process of the block $p_i = B.p$. The process states in the original protocol expect to receive indications from o, so these indications should be passed to $B.PS$ when interpreting B. When o returns an indication to p_i, it is the responsibility of p_i to attach the indications to the block in a special buffer $B.buff[\mathsf{o}]$. The contents of $B.buff[\mathsf{o}]$ are passed to $B.PS$ when interpreting B. This concludes the high level description of block interpretation. In order to interpret an entire block DAG, we interpret blocks in a topological order since the interpretation of each block B depends on the interpretation of its predecessors. Since the graph is a DAG, such an order exists and every block can be interpreted. The full algorithm interpret($\mathcal{G}, \mathcal{P}$) is presented in Algorithm 2. The main guarantee of interpret($\mathcal{G}, \mathcal{P}$) is the fact that the interpretation of B is independent of $\mathcal{G}$. This is formalized in the following lemma (proved in the full version [3]):

▶ **Lemma 3.** *For any two block DAGs G_1 and G_2, if $B \in G_1$ and $B \in G_2$ then the interpretation of B in both* interpret($G_1, \mathcal{P}$) *and* interpret($G_2, \mathcal{P}$) *is identical.*

Algorithm 2 interpret$(G_i, \mathcal{P})$ for process p_i.

```
$G_i = (V_i, E_i)$ is a block DAG and $\mathcal{P}$ is a public-coin protocol.
$G_i$ is process-local variable that maintains its value across different invocations
while $\exists B \in G_i$ s.t. $B$ is not interpreted s.t. $\forall B' \in B.preds : B'$ is interpreted do
    if $B.k = 0$ then
        Initialize $B.PS$ as a new state according to the protocol $\mathcal{P}$ and process $B.p$
    else
        $B.PS := B.parent.PS$
    for all $B' \in B.preds$ do
        Copy messages from $B'.M_{out}[B.p]$ to $B.M_{in}$
    Pass the user requests $B.rqsts$, messages $B.M_{in}$, random tape $B.rand$ and the object
  indications $B.buff$ to the state $B.PS$
    Overwrite the new state in $B.PS$
    Store the outgoing messages in $B.M_{out}$
    if $B.p = i$ then
        Return user indications produced by $B.PS$ to the user
        Perform object invocations as dictated by $B.PS$
```

Joint Block DAG. We now explain how processes build and maintain the block DAGs.

Algorithm 3 presents the genBlock(G_i) algorithm, which allows a process to generate blocks and inject data into the system. The algorithm gets a valid block DAG G_i of p_i. It then generates a new block B and assigns it a parent in G_i, then adds to $B.preds$ all references to blocks in G_i that do not have a path to $B.parent$. Note that since $B.preds \subseteq V_i$, then $B.pred$ only includes blocks B' s.t. $valid(p_i, B')$. This guarantees that B is a valid block. Next the external data is filled into the block: this includes moving the user requests from $Rqsts_i$ to $B.rqsts$, moving the object indications from $\mathsf{o}.buff_i$ to $B.buff[\mathsf{o}]$ for each relevant $\mathsf{o} \in \mathcal{O}$ and finally assigning a random string ρ to $B.rand$. Note that we do not know exactly how long ρ needs to be until B is actually interpreted. Since all $B' \in B.preds$ are already in G_i, process p_i can already interpret B and generate ρ while generating B.

Next, we describe the communication component that is responsible for exchanging blocks and growing the DAGs. We have shown that processes that interpret the same blocks reach the same conclusion. But for this to be useful, the communication component must ensure correct processes eventually interpret the same blocks. That is, if a correct process p_i adds some B to G_i, then every correct process p_j eventually adds B to G_j. This can be viewed as a consistency property between two processes.

Note that a naive approach of having each process simply send its blocks to everyone does not guarantee consistency, since an honest process p_i may add a block B^* by some corrupted process B^* as a predecessor for its own block B. p_i naturally considers B valid and adds it to its block DAG, but for any other honest process p_j, B will never be considered valid until it receives B^* from p^*.

Consistency can be achieved with the following simple *echoing* mechanism. For each block B that p_i issues using genBlock, p_i generates a signature for B which we denote by $B.\sigma$, and sends $(B, B.\sigma)$ to everyone. When p_i receives a block B by some other process, it first ensures B is authentic (by verifying the signature). After collecting all authentic blocks, p_i tries to validate as many of them as possible. The validation fails only if some $B' \in B.preds$ of B is missing, so p_i requests B' from the process $B.p$ that issued B, using a forward request message which we denote by $\mathsf{FWD}(\mathsf{ref}(B'))$. The idea is that if $B.p$ is correct

Algorithm 3 genBlock(G_i) for process p_i.

```
$G_i = (V_i, E_i)$ is a block DAG.
Initialize a new block $B$ as follows $B.p := p_i, B.preds := \emptyset, B.rqsts := \emptyset$
Assign to $B.parent$ the reference of the most recent block in $G_i$ issued by $p_i$.
$B.k := B.parent.k + 1$
for all $B' \in V_i$ s.t. $\neg\mathsf{path}(B', B.parent)$ do
    $B.preds := B.preds \cup \{\mathsf{ref}(B')\}$
Fill the external data $\mathsf{fillData}(B)$.
return $B$
```

Algorithm 4 exchangeBlocks(G_i) for process p_i.

```
$G_i = (V_i, E_i)$: a block DAG
$toValidate$, $isSent$: process-local variables, maintain their values across invocations
Initialize $toValidate := \emptyset$ and $isSent := \emptyset$
for all $B \in G_i$ s.t. $B.p = p_i$ and $B \notin isSent$ do
    Sign $B$ and denote the signature by $B.\sigma$
    Send $(B, B.\sigma)$ to everyone
Move all authentic blocks from all $In_{j\to i}$ to a set $auth$
$toValidate := toValidate \cup auth$                          ▷ Throw inauthentic blocks
while $\exists B \in toValidate$ s.t. $\mathsf{valid}(p_i, B)$ do
    $G_i.\mathsf{insert}(B)$
    $toValidate := toValidate \setminus \{B\}$
    $auth := auth \setminus \{B\}$
for all $B \in auth$ do                               ▷ Try to validate all authentic blocks
    for all $B' \in B.preds$ s.t. $B' \notin G_i$ do
        Send $\mathsf{FWD}(\mathsf{ref}(B'))$ to $B.p$          ▷ Request missing blocks from $B.p$
for all $\mathsf{FWD}(\mathsf{ref}(B'))$ in some $In_{j\to i}$ do          ▷ Respond to forward requests
    If $B' \in G_i$, send $(B', B'.\sigma)$ to $p_j$
Empty all $In_{j\to i}$.
```

then it must have those blocks, so it will eventually send them to p_i, allowing p_i to validate the block $B.p$. Finally, p_i of course has to respond to the forward requests it has received. The consistency guarantee ensured by exchangeBlocks is formalized in the following lemma:

▶ **Lemma 4.** *For any two correct processes p_i and p_j executing the protocol of Figure 2, if p_i adds a block to its block DAG G_i, then p_j eventually inserts B into G_j.*

We note that Lemma 4 really refer to any protocol in which Algorithms 3 and 4 are continuously run, and are not specific to Figure 2. The proof is deferred to the full version [3].

Correctness Proof. Combining Lemma 4 with Lemma 3 and assuming eventual delivery of blocks, we get eventual delivery of simulated messages. In other words, if a correct process p_i wants to send a message m to some correct process p_j, then this is expressed in the block DAG framework as a block B issued by p_i, such that $B.M_{out}[j]$ contains the message m. Delivering the message m to p_j is expressed by p_j creating a block B' such that $m \in B'.M_{in}$. Note that referring to unambiguous interpretations of B and B' is only possible through Lemma 3. By Lemma 4, we know that if p_i issues the block B then p_j eventually receives

B and considers it valid. By the algorithm in Algorithm 3, eventually p_j creates a new block B' such that $B \in B'.preds$ and by Algorithm 2, m will be added to $B.M_{in}$. This discussion demonstrates that the block DAG framework guarantees eventual delivery of simulated messages, if we assume eventual delivery of blocks. This guarantees the liveness of the block DAG simulation.

We show that the block DAG simulation of a protocol $\mathcal{P}$ is faithful in the sense that there exists a *forward simulation* from the block DAG simulation denoted as $\mathsf{BD}(\mathcal{P})$ to $\mathcal{P}$ (modeled as LTSs). As mentioned after 1, this implies that the block DAG simulation inherits finite-trace probability distributions of $\mathcal{P}$ and that typical specifications of programs using the DAG simulation instead of $\mathcal{P}$ are preserved.

Section 3 describes the modeling of $\mathcal{P}$ using LTSs. We describe below a modeling of $\mathsf{BD}(\mathcal{P})$ using an LTS $L' = (Q', \Sigma', q'_{start}, \delta')$ which simplifies the forward simulation proof. A state $q' \in Q'$ contains the block DAG G_i of each process p_i and $(In^B_{j\to i})_{j\in[n]}$ and $(Out^B_{i\to j})_{j\in[n]}$ for each process p_i, where $In^B_{j\to i}$ is the incoming buffer of process i with blocks sent by process j and $Out^B_{i\to j}$ is the outgoing buffer with blocks sent by i to j. As before, we assume that incoming user requests are stored in $In^B_{i\to i}$ and outgoing user indications are stored in $Out^B_{i\to i}$. The shared object indications are stored in separate buffers $(\mathsf{o}.\mathit{buff}_i)_{\mathsf{o}\in\mathcal{O}}$ as before. Overall, $q' = \big(G_i, (In^B_{j\to i})_{j\in[n]}, (Out^B_{i\to j})_{j\in[n]} (\mathsf{o}.\mathit{buff}_i)_{\mathsf{o}\in\mathcal{O}}\big)_{i\in[n]}$. In the initial state q'_{start}, all of the block DAGs and the buffers are empty. The transition labels correspond to computing and validating blocks, exchanging blocks, and user requests or indications. In comparison to the "standard" model described in Section 3 we decompose a compute step of a process as defined in Figure 2 into a sequence of steps. This simplifies the forward simulation proof. As before, we include the randomness (that is attached to the newly created block) in the computation label. Formally, the transition labels are as follows:

1. $\mathsf{validateBlock}(i \to j)$ denotes a transition where p_j validates a block issued by p_i (inside the $\mathsf{genBlock}$ algorithm).
2. $\mathsf{compute}(i, \rho)$ denotes a transition where process p_i produces and disseminates a new block (inside the $\mathsf{genBlock}$ algorithm) with ρ as its randomness, and then runs $\mathsf{interpret}$ to interpret the new block (and other previously uninterpreted blocks).
3. $\mathsf{sendFWD}(i \to j)$ denotes a transition where p_i sends a FWD request to p_j.
4. $\mathsf{replyFWD}(i \to j)$ denotes a transition where p_i sends a reply to a FWD sent by p_j.
5. $\mathsf{deliverBlocks}(i \to j)$ is a transition where all the blocks in $Out^B_{i\to j}$ are moved to $In^B_{i\to j}$.
6. $\mathsf{o.indicate}(i, w)$ denotes a transition where the value w has been added to $\mathsf{o}.\mathit{buff}_i$.
7. labels for user requests ($\mathsf{request}(i, x)$) or indications ($\mathsf{indicate}(i, y)$) are used as in Section 3.

The external actions Σ_E are defined exactly as for the LTS L modeling $\mathcal{P}$, presented in Section 3 (Σ_E includes $\mathsf{request}(i, x)$, $\mathsf{indicate}(i, y)$, and $\mathsf{compute}(i, \rho)$). A transition $(q'_1, e, q'_2) \in Q' \times \Sigma' \times Q'$ (denoted $q'_1 \xrightarrow{e} q'_2$) is in δ' if and only if the protocol $\mathsf{BD}(\mathcal{P})$ can get from state q'_1 to state q'_2 by executing the step denoted by the label e. Theorem 5 is proved in the full version [3].

▶ **Theorem 5.** *There exists a forward simulation from the LTS L' modeling* $\mathsf{BD}(\mathcal{P})$ *to the LTS L modeling $\mathcal{P}$.*

7 Relation to Prior Work

Comparison with the deterministic simulation. We can now discuss how our simulation and proof are related to the work of Schett and Danezis [17]. They show how block DAGs can be used to simulate deterministic protocols, which are a special case of the protocols that we handle here. Readers that are familiar with their work will notice that we were able to achieve a simulation that is a natural extension of theirs. We emphasize, however, that our techniques for proving the faithfulness of our simulation are novel and different from theirs. This is necessary because their techniques do not capture the probabilistic guarantees of randomized protocols.

Our network component which consists of genBlock and exchangeBlocks algorithms is a natural extension of the gossip algorithm of [17]. Indeed, the code responsible for generating new blocks and echoing them is almost identical to that of gossip. The difference is that because we want to exchange only blocks, they should carry enough information to resolve the randomized decisions that can come from local randomness or shared objects. In our protocol, each process is responsible to pass along its local randomness or the indications it got from the shared object in the blocks that it creates. Lemma 4 is proved in a manner similar to [17, Lemma 3.7].

Our interpretation algorithm is the natural extension of interpret algorithm of [17] for our context. That is, when interpreting a deterministic protocol, the computation of each process is only determined by the incoming messages and its state prior to processing those messages. When interpreting a randomized protocol with shared objects, the local computation may depend on local randomness and object indications. Our interpretation algorithm used those fields that were already attached to each block by our genBlock. Lemma 3 that states the common interpretation of block DAGs, is analogous to [17, Lemma 4.2]. However, the proof of the latter had a minor mistake and our proof is slightly different.

Finally, the guarantees of randomized protocols, unlike those of deterministic protocols, cannot always be expressed as trace properties. Particularly, for our simulation to be faithful to the original protocol, we need a more careful and precise statement and proof. Therefore, the modeling in Sections 3 and 6 as well as the proof of Theorem 5 are totally different from what appears in [17].

Analyzing existing protocols. Several recent works rely on the block DAG approach, e.g., Aleph [10], DAG-Rider [13] and Bullshark [20]. All of these protocols are randomized. While each of these works presents a new protocol, we provide a formal and systematic framework for analyzing DAG-based protocols, especially *randomized* block DAG protocols.

Here we discuss how our simulation applies to existing protocols, concentrating on Aleph [10] and DAG-Rider [13]. These protocols aim to order the blocks of the DAG, so as to implement *Byzantine Atomic Broadcast* (BAB). A BAB protocol allows all processes to receive the same messages in the *same order*. One natural way of implementing a BAB protocol using a block DAG is by having each process attach the messages it wants to broadcast to a block and then broadcast the block to everyone. The processes then just need to agree on an order of the blocks, which would induce an order of the messages. Like our simulation, both Aleph and DAG-Rider have a communication component that is responsible for building and maintaining the common DAG. In both protocols, each block in the DAG belongs to a specific round, and each correct process has a single block in each round.

Aleph orders the blocks in the DAG by electing a leader block in each round, and then having that leader block (deterministically) dictate the order of its ancestor blocks that have not been ordered yet.

DAG-Rider divides the DAG into *waves*. Each wave consists of four consecutive rounds, and a leader block is elected for each wave. The block leader election is done by interpreting the (same) block DAG as a consensus protocol and utilizing a shared object for generating randomness, namely, a common coin. It is critical to note that our simulation preserves the properties of the shared object, for example the *unpredictability* of the common coin. This is because our forward simulation preserves the compute events, in which the object invocations happen. This means that the object cannot distinguish if it is being used in the context of the original protocol or in the context of the block DAG simulation of the protocol. This means that its properties are preserved.

Aleph and DAG-Rider can be analyzed using our framework. The consensus protocol used can be analyzed independently of Aleph or DAG-Rider, while assuming it has access to a common coin. By Theorem 5, the simulation of the consensus protocol on the block DAG is faithful to the original consensus protocol. This not only simplifies reasoning about safety and liveness of Aleph and DAG-Rider, but also supports *modularity*: the simulated consensus protocol in Aleph or DAG-Rider can be seamlessly replaced using Theorem 5.

8 Discussion

We have presented a faithful simulation of DAG-based BFT protocols, which use public coins and shared objects, including protocols that utilize a common source of randomness, e.g., a *common coin*. Being faithful, the simulation precisely preserves properties of the original BFT protocol, and in particular, their probability distributions.

One of the appealing properties of our block DAG framework is that it allows to minimize the communication when running multiple instances of potentially different protocols. This can be done by using the same joint block DAG to interpret multiple protocol instances. The logic of the communication layer does not change, other than the need to specify the associated instance for each user request and object indication that is attached to the blocks. Each process would then run multiple interpretation instances, one for each protocol instance. We note that a process does not necessarily need to attach a separate randomness tape for each instance, and can instead attach a small random seed. Processes can then use a *pseudorandom generator* to expand the seed to a large enough pseudorandom string that can be used for all of the instances. This ensures that block size does not grow beyond the size of the user requests and the object indications.

Our simulation relies on the fact that it is safe to reveal the randomness to the adversary as soon as it is used. We can similarly define *private-coin* protocols, whose security relies on processes ability to keep secrets from the adversary. A classical example would be any Asynchronous Verifiable Secret Sharing scheme (e.g. [5]). From a theoretical point of view, it would be interesting to demonstrate how we can simulate such algorithms on block DAGs. However, we note that some protocols are entirely public-coin other than a dedicated private-coin sub-protocol, such as Aleph-Beacon in Aleph [10] (which is used to implement a common coin). In this case, the dedicated sub-protocol can be encapsulated as a shared object, thus factoring out the use of private-coin simulations.

References

1 Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25–29, 2022*, pages 381–391. ACM, 2022. doi:10.1145/3519270.3538426.

2 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programsthat use concurrent objects. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.2.

3 Hagit Attiya, Constantin Enea, and Shafik Nassar. Faithful simulation of randomized bft protocols on block dags. Cryptology ePrint Archive, Paper 2023/192, 2023. URL: `https://eprint.iacr.org/2023/192`.

4 Leemon Baird. The Swirlds Hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. `https://www.researchhub.com/paper/337/the-swirlds-hashgraph-consensus-algorithm-fair-fast-byzantine-fault-tolerance`, 2016.

5 Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 42–51. ACM, 1993. `doi:10.1145/167088.167105`.

6 Anton Churyumov. Byteball: A decentralized system for storage and transfer of value. `https://byteball.org/Byteball.pdf`, 2016.

7 Danny Dolev. The Byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982. `doi:10.1016/0196-6774(82)90004-9`.

8 Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Weak progressive forward simulation is necessary and sufficient for strong observational refinement. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 31:1–31:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CONCUR.2022.31`.

9 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. In Ronald Fagin and Philip A. Bernstein, editors, *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*, pages 1–7. ACM, 1983. `doi:10.1145/588058.588060`.

10 Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. Aleph: Efficient atomic broadcast in asynchronous networks with Byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, pages 214–228. ACM, 2019. `doi:10.1145/3318041.3355467`.

11 Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382. ACM, 2011. `doi:10.1145/1993636.1993687`.

12 Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement in polynomial time with near-optimal resilience. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20–24, 2022*, pages 502–514. ACM, 2022. `doi:10.1145/3519935.3520015`.

13 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021. `doi:10.1145/3465084.3467905`.

14 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. `doi:10.1145/359545.359563`.

15 Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. `doi:10.1006/inco.1995.1134`.

16 Sean Rowan and Naïri Usher. The Flare consensus protocol: Fair, fast federated Byzantine agreement consensus. `https://flareportal.com/wp-content/uploads/simple-file-list/FCP.pdf`, 2019.

17 Maria Anna Schett and George Danezis. Embedding a deterministic BFT protocol in a block DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 177–186. ACM, 2021. `doi:10.1145/3465084.3467930`.

18 Roberto Segala. A compositional trace-based semantics for probabilistic automata. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, volume 962 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 1995. doi:10.1007/3-540-60218-6_17.

19 Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. PHANTOM GHOSTDAG: a scalable generalization of nakamoto consensus: September 2, 2021. In Foteini Baldimtsi and Tim Roughgarden, editors, *AFT '21: 3rd ACM Conference on Advances in Financial Technologies, Arlington, Virginia, USA, September 26–28, 2021*, pages 57–70. ACM, 2021. doi:10.1145/3479722.3480990.

20 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022. doi:10.1145/3548606.3559361.

21 Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. SoK: Diving into DAG-based blockchain systems. *CoRR*, abs/2012.06128, 2020. arXiv:2012.06128.

Real Equation Systems with Alternating Fixed-Points

Jan Friso Groote
Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands

Tim A. C. Willemse
Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands

Abstract

We introduce the notion of a Real Equation System (RES), which lifts Boolean Equation Systems (BESs) to the domain of extended real numbers. Our RESs allow arbitrary nesting of least and greatest fixed-point operators. We show that each RES can be rewritten into an equivalent RES in normal form. These normal forms provide the basis for a complete procedure to solve RESs. This employs the elimination of the fixed-point variable at the left side of an equation from its right-hand side, combined with a technique often referred to as Gauß-elimination. We illustrate how this framework can be used to verify quantitative modal formulas with alternating fixed-point operators interpreted over probabilistic labelled transition systems.

2012 ACM Subject Classification Theory of computation → Modal and temporal logics; Theory of computation → Verification by model checking

Keywords and phrases Real Equation System, Solution method, Gauß-elimination, Model checking, Quantitative modal mu-calculus

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.28

Related Version *Full Version*: https://arxiv.org/abs/2307.07455

1 Introduction

The modal mu-calculus is a logic that allows to formulate and verify a very wide range of properties on behaviour, far more expressive than virtually any other behavioural logic around [3, 2]. For instance, CTL and LTL can be mapped to it, but the reverse is not possible. By allowing data parameters in the fixed point variables in modal formulas, this can even be done linearly, without loss of computational effectiveness [5]. Using alternating fixed-points, the modal mu-calculus can intrinsically express various forms of fairness, which in other logics can often only be achieved by adding special fairness operators.

An effective way to evaluate a modal property on a labelled transition system is by translating both to a single Boolean Equation System (BES) with alternating fixed-points [20, 22]. Exactly if the initial boolean variable of the obtained BES has the solution true, the property is valid for the labelled transition system. A BES with alternating fixed-points is equivalent to a parity game [21, 2]. There are many algorithms to solve BESs and parity games [26, 4, 17, 25]. Although, it is a long standing open problem whether a polynomial algorithm exists to solve BESs [4, 17], the existing algorithms work remarkably well in practical contexts.

For a while now, it has been argued that modal logics can become even more effective if they provide quantitative answers [15, 16], such as durations, probabilities and expected values. In this paper we lift boolean equation systems to real numbers to form a framework for the evaluation of quantitative modal formulas, and call the result *Real Equation Systems* (*RESs*), i.e., fixed-point equation systems over the domain of the extended reals, $\mathbb{R} \cup \{-\infty, \infty\}$.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 28; pp. 28:1–28:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Conjunction and disjunction are interpreted as minimum and maximum, and new operators such as addition and multiplication with positive constants are added. A typical example of a real equation system is the following

$$\begin{aligned}&\mu X = (\tfrac{1}{2}X + 1) \vee (\tfrac{1}{5}Y + 3),\\ &\nu Y = ((\tfrac{1}{10}Y - 10) \vee (2X + 5)) \wedge 17.\end{aligned}$$

Based on Tarski's fixed-point theorem, this real equation system has a unique solution. Using the method provided in this paper we can determine this solution using algebraic manipulation. In the case above, see Section 4, the second fixed-point equation can be simplified to $\nu Y = -\frac{100}{9} \vee ((2X + 5) \wedge 17)$. It is sound to substitute this in the first equation, which becomes $\mu X = (\frac{1}{2}X + 1) \vee \frac{7}{9} \vee ((\frac{2}{5}X + 4) \wedge \frac{32}{5})$. This equation can be solved for X yielding $X = \frac{32}{5}$, from which it directly follows that $Y = 17$.

Concretely, this paper has the following results. We define real equation systems with alternating fixed-points. The base syntax for expressions is equal to that of [7] with constants, minimum, maximum, addition and multiplication with positive real constants. We add four additional operators, namely two conditional operators, and two tests for infinity, which turn out to be required to algebraically solve arbitrary real equation systems.

We provide algebraic laws that allow to transform any expression to *conjunctive/disjunctive normal form*. Based on this normal form we provide rules that allow to eliminate each variable bound in the left-hand side of an equation from the right-hand side of that equation. This enables "Gauß-elimination", developed for BESs, using which any real equation system can be solved.

We provide a quantitative modal logic, and define how a quantitative formula and a (probabilistic) labelled transition system ((p)LTS) can be transformed into a RES. The solution of the initial variable of this equation system is equal to the evaluation of the quantitative formula on the labelled transition system. We also briefly touch upon the embedding of BESs into RESs.

The approach in this paper follows the tradition of boolean equation systems [19, 20, 21]. By allowing data parameters in the fixed-point variables we obtain Parameterised Boolean Equation Systems (PBESs) which is a very expressive framework that forms the workhorse for model checking [22, 13, 11]. In this paper we do not address such parametric extensions, as they are pretty straightforward, but in combination with parameterised quantitative modal logic, it will certainly provide a very versatile framework for quantitative model checking.

There are a number of extensions of the boolean equation framework to the setting of reals but these typically limit themselves to only single fixed-points. In [7] the minimal integer solutions for a set of equations with only minimal fixed-points is determined. In [8] a polynomial algorithm is provided to find the minimal solution for a set of real equation systems. In [1] convex lattice equation systems are introduced, also restricted to a single fixed-point. In that paper a proof system is given to show that all models of the equations are consistent, meaning that the evaluation of a quantitative modal formula is limited by some upper-bound.

In [24], the Łukasiewicz μ-calculus is studied, which resembles RESs restricted to the interval $[0, 1]$. This logic does allow minimal and maximal fixed-points. They provide two algorithmic ways of computing the solutions for formulas in their logic, *viz.* an indirect method that builds formulas in the first-order theory of linear arithmetic and exploits quantifier elimination, and a method that uses iteration to refine successive approximations of conditioned linear expressions. Embedding our logic in the Łukasiewicz μ-calculus can be done by mapping the extended reals onto the interval $[0, 1]$ using an appropriate sigmoid function. But such a mapping does not map our addition and constant multiplication to available counterparts in the Łukasiewicz μ-calculus, which prevents using algorithms for Łukasiewicz μ-terms [18, 24] to our setting. However, as the Łukasiewicz μ-calculus is directly encodable into the RES framework, all our results are directly applicable to the Łukasiewicz μ-calculus.

2 Expressions and normal forms

We work in the setting of *extended real numbers*, i.e., $\mathbb{R}\cup\{\infty,-\infty\}$, denoted by $\hat{\mathbb{R}}$. We assume the normal total ordering $\leq$ on $\hat{\mathbb{R}}$ where $-\infty \leq x$ and $x \leq \infty$ for all $x \in \hat{\mathbb{R}}$. Throughout this text we employ a set $\mathcal{X}$ of variables and *valuations* $\eta : \mathcal{X} \to \hat{\mathbb{R}}$ that map variables to extended reals. We write $\eta(X)$ to apply η to X, and $\eta[X := r]$ to adapt valuations by:

$$\eta[X := r](Y) = \begin{cases} r & \text{if } X = Y, \\ \eta(Y) & \text{otherwise.} \end{cases}$$

We consider expressions over the set $\mathcal{X}$ of variables with the following syntax.

$$e \ ::= \ X \mid d \mid c{\cdot}e \mid e+e \mid e\wedge e \mid e\vee e \mid e \Rightarrow e \diamond e \mid e \to e \diamond e \mid eq_{\infty}(e) \mid eq_{-\infty}(e)$$

where $X \in \mathcal{X}$, $d \in \hat{\mathbb{R}}$ is a constant, $c \in \mathbb{R}_{>0}$ a positive constant, $+$ represents addition, $\wedge$ stands for minimum, $\vee$ for maximum, $\_ \Rightarrow \_ \diamond \_$ and $\_ \to \_ \diamond \_$ are conditional operators, and eq_{∞} and $eq_{-\infty}$ are auxiliary functions to check for $\pm\infty$. The conditional operators and the checks for infinity occur naturally while solving fixed-point equations and therefore, we made them part of the syntax. We apply valuations to expressions, as in $\eta(e)$, where η distributes over all operators in the expression.

The interpretation of these operators on the domain $\hat{\mathbb{R}}$ is largely obvious. A variable X gets a value by a valuation. Multiplying expressions with a constant c is standard, and yields $\pm\infty$ if applied on $\pm\infty$. The conditional operators, addition and infinity operators are defined below where $e, e_1, e_2, e_3 \in \hat{\mathbb{R}}$.

$$e_1 + e_2 = \begin{cases} e_1 + e_2 & \text{if } e_1, e_2 \in \mathbb{R}, \text{ i.e., apply normal addition,} \\ \infty & \text{if } e_1 = \infty \text{ or } e_2 = \infty, \\ -\infty & \text{if } e_i{=}{-\infty} \text{ and } e_{3-i} \neq \infty \text{ for } i = 1, 2. \end{cases}$$

$$e_1 \Rightarrow e_2 \diamond e_3 = \begin{cases} e_2 \wedge e_3 & \text{if } e_1 \leq 0, \\ e_3 & \text{if } e_1 > 0. \end{cases} \qquad e_1 \to e_2 \diamond e_3 = \begin{cases} e_2 & \text{if } e_1 < 0, \\ e_2 \vee e_3 & \text{if } e_1 \geq 0. \end{cases}$$

$$eq_{\infty}(e) = \begin{cases} \infty & \text{if } e = \infty, \\ -\infty & \text{if } e \neq \infty. \end{cases} \qquad eq_{-\infty}(e) = \begin{cases} \infty & \text{if } e \neq -\infty, \\ -\infty & \text{if } e = -\infty. \end{cases}$$

Note that all defined operators are monotonic on $\hat{\mathbb{R}}$. We have the identity $eq_{\infty}(e) = e + -\infty$, and so, we do not treat eq_{∞} as a primary operator. We write $e[X := e']$ for the expression representing the syntactic substitution of e' for X in e. We write $\mathsf{occ}(e)$ for the set of variables from $\mathcal{X}$ occurring in e. Table 1 contains many useful algebraic laws for our operators.

The addition operator $+$ has as property that $-\infty + \infty = \infty + -\infty = \infty$. One may require the other natural addition operator $\hat{+}$, as used in [8], satisfying that $-\infty\hat{+}\infty = \infty\hat{+} - \infty = -\infty$. It can be defined as follows:

$$e_1 \hat{+} e_2 \ = \ eq_{-\infty}(e_1) \Rightarrow -\infty \diamond (eq_{-\infty}(e_2) \Rightarrow -\infty \diamond (e_1 + e_2)).$$

We can extend the syntax with unary negation $-e$ with its standard meaning, and, provided no variable occurs in the scope of its definition within an odd number of negations, negation can be eliminated using standard simplification rules. Therefore, we do not consider it as a primary part of our syntax. At the end of Table 1 we list several identities involving negation. Note that operators $+$ and $\hat{+}$ are each other's dual with regard to negation.

We introduce normal forms, crucial to solve real equation systems, where the sum, conjunction and disjunction over empty domains of variables equal 0, ∞ and $-\infty$, respectively.

▶ **Definition 1.** *Let $\mathcal{X}$ be a set of variables. An expression e is in simple conjunctive normal form iff it has the shape*

$$\bigwedge_{i \in I} \bigvee_{j \in J_i} \big(\big(\sum_{X \in \mathcal{X}_{ij}} c_{ij}^{X} \cdot X\big) + \big(\sum_{X \in \mathcal{X}'_{ij}} eq_{-\infty}(X)\big) + d_{ij}\big)$$

and it is in simple disjunctive normal form iff it has the shape

$$\bigvee_{i \in I} \bigwedge_{j \in J_i} \big(\big(\sum_{X \in \mathcal{X}_{ij}} c_{ij}^{X} \cdot X\big) + \big(\sum_{X \in \mathcal{X}'_{ij}} eq_{-\infty}(X)\big) + d_{ij}\big)$$

where $\mathcal{X}_{ij} \subseteq \mathcal{X}$ and $\mathcal{X}'_{ij} \subseteq \mathcal{X}$ are finite sets of variables, $c_{ij}^{X} \in \mathbb{R}_{>0}$, and $d_{ij} \in \hat{\mathbb{R}}$.

An expression e is in conjunctive, resp. disjunctive normal form iff

1. *e is in simple conjunctive, resp. disjunctive normal form, or*
2. *e has the shape $e_1 \Rightarrow e_2 \diamond e_3$ or $e_1 \rightarrow e_2 \diamond e_3$ where e_1 is in simple conjunctive, resp. disjunctive normal form and e_2 and e_3 are conjunctive resp. disjunctive normal forms.*

▶ **Lemma 2.** *Each expression e not containing the conditional operators $e_1 \Rightarrow e_2 \diamond e_3$ or $e_1 \rightarrow e_2 \diamond e_3$ can be rewritten to a simple conjunctive or disjunctive normal form using the equations in Table 1.*

▶ **Lemma 3.** *Expression of the forms $e_1 \Rightarrow e_2 \diamond e_3$ and $e_1 \rightarrow e_2 \diamond e_3$ can be rewritten to equivalent expressions where the first argument of such a conditional operator is a simple conjunctive or disjunctive normal form using the equations in Table 1.*

▶ **Theorem 4.** *Each expression e can be rewritten to both a conjunctive and a disjunctive normal form using the equations in Table 1.*

3 Real equation systems and Gauß-elimination

In this section we introduce Real Equation Systems (RESs) as sequences of fixed-point equations, introduce a natural equivalence between RESs, and provide a generic solution method, known as Gauß-elimination [20].

▶ **Definition 5.** *Let $\mathcal{X}$ be a set of variables. A Real Equation System (RES) $\mathcal{E}$ is a finite sequence of (fixed-point) equations*

$$\sigma_1 X_1 {=} e_1, \ldots, \sigma_n X_n {=} e_n$$

where σ_i is either the minimal fixed-point operator μ or the maximal fixed-point operator ν, $X_i \in \mathcal{X}$ are variables and e_i are expressions. We write $\mathsf{bnd}(\mathcal{E})$ for the set of variables occurring in the left-hand side, i.e., $\mathsf{bnd}(\mathcal{E}) = \{X_1, \ldots, X_n\}$.

Table 1 Algebraic laws.

$\mathrm{I}_{\vee}$	$e \vee e = e$	$\mathrm{I}_{\wedge}$	$e \wedge e = e$
D_{+}^{+}	$(e_1 + e_2) + e_3 = e_1 + (e_2 + e_3)$	$\mathrm{C}+$	$e_1 + e_2 = e_2 + e_1$
$\mathrm{D}_{\vee}^{\vee}$	$(e_1 \vee_2) \vee e_3 = e_1 \vee (e_2 \vee e_3)$	$\mathrm{C}\vee$	$e_1 \vee e_2 = e_2 \vee e_1$
$\mathrm{D}_{\wedge}^{\wedge}$	$(e_1 \wedge e_2) \wedge e_3 = e_1 \wedge (e_2 \wedge e_3)$	$\mathrm{C}\wedge$	$e_1 \wedge e_2 = e_2 \wedge e_1$
$\mathrm{D}_{\Rightarrow}^{\Rightarrow}$	$(e_1 \Rightarrow e_2 \diamond e_3) \Rightarrow f_1 \diamond f_2 = ((e_1 \vee e_2) \wedge e_3) \Rightarrow f_1 \diamond f_2$		
$\mathrm{D}_{\rightarrow}^{\Rightarrow}$	$(e_1 \Rightarrow e_2 \diamond e_3) \rightarrow f_1 \diamond f_2 = e_1 \rightarrow (e_2 \Rightarrow f_1 \diamond f_2) \diamond (e_2 \vee e_3 \Rightarrow f_1 \diamond f_2)$		
$\mathrm{D}_{\Rightarrow}^{c}$	$c{\cdot}(e_1 \Rightarrow e_2 \diamond e_3) = e_1 \Rightarrow c{\cdot}e_2 \diamond c{\cdot}e_3$		
$\mathrm{D}_{\Rightarrow}^{+}$	$(e_1 \Rightarrow e_2 \diamond e_3) + f = e_1 \Rightarrow (e_2 + f) \diamond (e_3 + f)$		
$\mathrm{D}_{\Rightarrow}^{\wedge}$	$(e_1 \Rightarrow e_2 \diamond e_3) \wedge f = e_1 \Rightarrow (e_2 \wedge f) \diamond (e_3 \wedge f)$		
$\mathrm{D}_{\Rightarrow}^{\vee}$	$(e_1 \Rightarrow e_2 \diamond e_3) \vee f = e_1 \Rightarrow (e_2 \vee f) \diamond (e_3 \vee f)$		
$\mathrm{D}_{\rightarrow}^{\rightarrow}$	$(e_1 \rightarrow e_2 \diamond e_3) \rightarrow f_1 \diamond f_2 = (e_2 \vee (e_1 \wedge e_3)) \rightarrow f_1 \diamond f_2$		
$\mathrm{D}_{\rightarrow}^{\Rightarrow}$	$(e_1 \rightarrow e_2 \diamond e_3) \Rightarrow f_1 \diamond f_2 = e_1 \Rightarrow (e_2 \wedge e_3 \rightarrow f_1 \diamond f_2) \diamond (e_3 \rightarrow f_1 \diamond f_2)$		
$\mathrm{D}_{\rightarrow}^{c}$	$c{\cdot}(e_1 \rightarrow e_2 \diamond e_3) = e_1 \rightarrow c{\cdot}e_2 \diamond c{\cdot}e_3$		
$\mathrm{D}+$	$(e_1 \rightarrow e_2 \diamond e_3) + f = e_1 \rightarrow (e_2 + f) \diamond (e_3 + f)$		
$\mathrm{D}_{\rightarrow}^{\wedge}$	$(e_1 \rightarrow e_2 \diamond e_3) \wedge f = e_1 \rightarrow (e_2 \wedge f) \diamond (e_3 \wedge f)$		
$\mathrm{D}_{\rightarrow}^{\vee}$	$(e_1 \rightarrow e_2 \diamond e_3) \vee f = e_1 \rightarrow (e_2 \vee f) \diamond (e_3 \vee f)$		
$\mathrm{D}_{\wedge}^{+}$	$e_1 + (e_2 \wedge e_3) = (e_1 + e_2) \wedge (e_1 + e_3)$	$\mathrm{D}_{\vee}^{+}$	$e_1 + (e_2 \vee e_3) = (e_1 + e_2) \vee (e_1 + e_3)$
D_{+}^{c}	$c{\cdot}(e_1 + e_2) = c{\cdot}e_1 + c{\cdot}e_2$		
$\mathrm{D}_{\wedge}^{c}$	$c{\cdot}(e_1 \wedge e_2) = c{\cdot}e_1 \wedge c{\cdot}e_2$	$\mathrm{D}_{\vee}^{c}$	$c{\cdot}(e_1 \vee e_2) = c{\cdot}e_1 \vee c{\cdot}e_2$
$\mathrm{D}_{\vee}^{\wedge}$	$e_1 \wedge (e_2 \vee e_3) = (e_1 \wedge e_2) \vee (e_1 \wedge e_3)$	$\mathrm{D}_{\wedge}^{\vee}$	$e_1 \vee (e_2 \wedge e_3) = (e_1 \vee e_2) \wedge (e_1 \vee e_3)$
$\mathrm{D}_{\infty}^{\infty}$	$eq_{\infty}(eq_{\infty}(e)) = eq_{\infty}(e)$	$\mathrm{D}_{\infty}^{-\infty}$	$eq_{-\infty}(eq_{\infty}(e)) = eq_{\infty}(e)$
$\mathrm{D}_{-\infty}^{\infty}$	$eq_{\infty}(eq_{-\infty}(e)) = eq_{-\infty}(e)$	$\mathrm{D}_{-\infty}^{-\infty}$	$eq_{-\infty}(eq_{-\infty}(e)) = eq_{-\infty}(e)$
D_{c}^{∞}	$eq_{\infty}(c{\cdot}e) = eq_{\infty}(e)$	$\mathrm{D}_{c}^{-\infty}$	$eq_{-\infty}(c{\cdot}x) = eq_{-\infty}(x)$
D_{+}^{∞}	$eq_{\infty}(e_1 + e_2) = eq_{\infty}(e_1) + eq_{\infty}(e_2) = eq_{\infty}(e_1) \vee eq_{\infty}(e_2)$		
$\mathrm{D}_{+}^{-\infty}$	$eq_{-\infty}(e_1 + e_2) = (eq_{-\infty}(e_1) \vee eq_{\infty}(e_2)) \wedge (eq_{\infty}(e_1) \vee eq_{-\infty}(e_2))$		
$\mathrm{D}_{\vee}^{\infty}$	$eq_{\infty}(e_1 \vee e_2) = eq_{\infty}(e_1) \vee eq_{\infty}(e_2)$	$\mathrm{D}_{\vee}^{-\infty}$	$eq_{-\infty}(e_1 \vee e_2) = eq_{-\infty}(e_1) \vee eq_{-\infty}(e_2)$
$\mathrm{D}_{\wedge}^{\infty}$	$eq_{\infty}(e_1 \wedge e_2) = eq_{\infty}(e_1) \wedge eq_{\infty}(e_2)$	$\mathrm{D}_{\wedge}^{-\infty}$	$eq_{-\infty}(e_1 \wedge e_2) = eq_{-\infty}(e_1) \wedge eq_{-\infty}(e_2)$
$\mathrm{E}_{\infty}^{\wedge}$	$eq_{\infty}(e) \wedge eq_{-\infty}(e) = eq_{\infty}(e)$	$\mathrm{E}_{-\infty}^{\vee}$	$eq_{\infty}(e) \vee eq_{-\infty}(e) = eq_{-\infty}(e)$
$\mathrm{D}_{\Rightarrow}^{\infty}$	$eq_{\infty}(e_1 \Rightarrow e_2 \diamond e_3) = e_1 \Rightarrow eq_{\infty}(e_2) \diamond eq_{\infty}(e_3)$		
$\mathrm{D}_{\Rightarrow}^{-\infty}$	$eq_{-\infty}(e_1 \Rightarrow e_2 \diamond e_3) = e_1 \Rightarrow eq_{-\infty}(e_2) \diamond eq_{-\infty}(e_3)$		
$\mathrm{D}_{\rightarrow}^{\infty}$	$eq_{\infty}(e_1 \rightarrow e_2 \diamond e_3) = e_1 \rightarrow eq_{\infty}(e_2) \diamond eq_{\infty}(e_3)$		
$\mathrm{D}_{\rightarrow}^{-\infty}$	$eq_{-\infty}(e_1 \rightarrow e_2 \diamond e_3) = e_1 \rightarrow eq_{-\infty}(e_2) \diamond eq_{-\infty}(e_3)$		
D_{c}^{-}	$-c{\cdot}e = c{\cdot} -e$		
D_{+}^{-}	$-(e_1 + e_2) = -e_1 \hat{+} -e_2$	$\mathrm{D}_{\hat{+}}^{-}$	$-(e_1 \hat{+} e_2) = -e_1 + -e_2$
$\mathrm{D}_{\vee}^{-}$	$-(e_1 \vee e_2) = -e_1 \wedge -e_2$	$\mathrm{D}_{\wedge}^{-}$	$-(e_1 \wedge e_2) = -e_1 \vee -e_2$
$\mathrm{D}_{\Rightarrow}^{-}$	$-(e_1 \Rightarrow e_2 \diamond e_3) = -e_1 \rightarrow -e_3 \diamond -e_2$	$\mathrm{D}_{\rightarrow}^{-}$	$-(e_1 \rightarrow e_2 \diamond e_3) = -e_1 \Rightarrow -e_3 \diamond -e_2$
D_{∞}^{-}	$-eq_{\infty}(e) = eq_{-\infty}(-e)$	$\mathrm{D}_{-\infty}^{-}$	$-eq_{-\infty}(e) = eq_{\infty}(-e)$

The empty sequence of equations is denoted by ε.

The semantics of a real equation system is a valuation giving the solutions of all variables, based on an initial valuation η giving the solution for all variables not bound in $\mathcal{E}$.

▶ **Definition 6.** *Let $\mathcal{X}$ be a set of variables and $\mathcal{E}$ be a real equation system over $\mathcal{X}$. The solution $[\![\mathcal{E}]\!]\eta : \mathcal{X} \to \hat{\mathbb{R}}$ yields an extended real number for all $X \in \mathcal{X}$, given a valuation $\eta : \mathcal{X} \to \hat{\mathbb{R}}$ of $\mathcal{E}$. It is inductively defined as follows:*

$$[\![\varepsilon]\!]\eta = \eta,$$
$$[\![\sigma X{=}e, \mathcal{E}]\!]\eta = [\![\mathcal{E}]\!](\eta[X := \sigma(X, \mathcal{E}, \eta, e)])$$

where $\sigma(X, \mathcal{E}, \eta, e)$ is defined as

$$\mu(X, \mathcal{E}, \eta, e) = \bigwedge\{r \in \hat{\mathbb{R}} \mid r \geq [\![\mathcal{E}]\!](\eta[X := r])(e)\} \text{ and}$$
$$\nu(X, \mathcal{E}, \eta, e) = \bigvee\{r \in \hat{\mathbb{R}} \mid [\![\mathcal{E}]\!](\eta[X := r])(e) \geq r\}.$$

It is equivalent to write $=$ instead of $\geq$ in the above sets. This makes the fixed-points easier to understand. Note that if the real equation system is closed, i.e., all variables in the right-hand sides occur in $\mathsf{bnd}(\mathcal{E})$, the value $[\![\mathcal{E}]\!]\eta(X)$ is independent of η for all $X \in \mathsf{bnd}(\mathcal{E})$.

Following [14], we introduce the notion of equivalency between equation systems. We use the symbol $\equiv$ to distinguish this equivalence from "$=$" used in equation systems.

▶ **Definition 7.** *Let $\mathcal{E}, \mathcal{E}'$ be real equation systems. We say that $\mathcal{E} \equiv \mathcal{E}'$ iff $[\![\mathcal{E}, \mathcal{F}]\!]\eta = [\![\mathcal{E}', \mathcal{F}]\!]\eta$ for all valuations η and real equation systems $\mathcal{F}$ with $\mathsf{bnd}(\mathcal{F}) \cap (\mathsf{bnd}(\mathcal{E}) \cup \mathsf{bnd}(\mathcal{E}')) = \emptyset$.*

In [14] it was observed that defining $\mathcal{E} \equiv \mathcal{E}'$ as $[\![\mathcal{E}]\!]\eta = [\![\mathcal{E}']\!]\eta$ for all η is not desirable, as the resulting equivalence is not a congruence. With this alternative notion, we find that $\mu X{=}Y$ and $\nu X{=}Y$ are equivalent. But $\mu X{=}Y, \nu Y{=}X$ and $\nu X{=}Y, \nu Y{=}X$ are not as the first one has solution $X = Y = -\infty$ and the second one has $X = Y = \infty$.

However, if the fixed-point symbol is the same, it is not necessary to take surrounding equations into account. This is a pretty useful lemma which makes the proofs in this paper much easier, and of which we are not aware that it occurs elsewhere in the literature.

▶ **Lemma 8.** *Let X be a variable, e and f be expressions and σ either the minimal or the maximal fixed-point symbol. If for any valuation η it holds that $[\![\sigma X = e]\!]\eta = [\![\sigma X = f]\!]\eta$ then $\sigma X = e \equiv \sigma X = f$.*

The proof of the main Theorem 11 is quite involved and heavily uses the following two lemmas, which we only give for the minimal fixed-point. The formulations for the maximal fixed-point are dual.

▶ **Lemma 9.** *Let $X \in \mathcal{X}$ be a variable and e, f be expressions. It holds that $\mu X = e \;\equiv\; \mu X = f$ if for every valuation η:*

1. *for the smallest $r \in \hat{\mathbb{R}}$ such that $r = \eta[X := r](e)$ it holds that there is an $r' \in \hat{\mathbb{R}}$ satisfying that $r' \leq r$ and $r' \geq \eta[X := r'](f)$, and, vice versa,*
2. *for the smallest $r \in \hat{\mathbb{R}}$ such that $r = \eta[X := r](f)$ it holds that there is an $r' \in \hat{\mathbb{R}}$ satisfying that $r' \leq r$ and $r' \geq \eta[X := r'](e)$.*

▶ **Lemma 10.** *If $\mu X = e \;\equiv\; \mu X = f$, then for any valuation η it holds that*

1. *for any $r \in \hat{\mathbb{R}}$ such that $r \geq \eta[X := r](e)$, there is an $r' \in \hat{\mathbb{R}}$ such that $r' \leq r$ and $r' = \eta[X := r'](f)$, and, vice versa,*
2. *for any $r \in \hat{\mathbb{R}}$ such that $r \geq \eta[X := r](f)$, there is an $r' \in \hat{\mathbb{R}}$ such that $r' \leq r$ and $r' = \eta[X := r'](e)$.*

Table 2 Properties of the equivalence $\equiv$ on RESs.

E1 $$\frac{\mathcal{E} \equiv \mathcal{E}'}{\mathcal{F}, \mathcal{E} \ \equiv\ \mathcal{F}, \mathcal{E}'}.$$

E2 $$\frac{\mathcal{E} \equiv \mathcal{E}'}{\mathcal{E}, \mathcal{F} \ \equiv\ \mathcal{E}', \mathcal{F}}.$$

E3 $\sigma X{=}e, \mathcal{E}, \sigma' Y{=}e' \ \equiv\ \sigma X{=}e[Y := e'], \mathcal{E}, \sigma' Y{=}e' \quad$ if $X, Y \notin \mathsf{bnd}(\mathcal{E})$.

E4 $\sigma X{=}e, \mathcal{E} \ \equiv\ \mathcal{E}, \sigma X{=}e \quad$ if $\mathsf{occ}(e) = \emptyset$ and $X \notin \mathsf{bnd}(\mathcal{E})$.

E5 $\sigma X{=}e, \sigma Y{=}e' \ \equiv\ \sigma Y{=}e', \sigma X{=}e$.

E6 $$\frac{\mu X = e_1 \ \equiv\ \mu X = f_1 \text{ and } \mu X = e_2 \ \equiv\ \mu X = f_2}{\mu X = e_1 \wedge e_2 \ \equiv\ \mu X = f_1 \wedge f_2}.$$

E7 $$\frac{\nu X = e_1 \ \equiv\ \nu X = f_1 \text{ and } \nu X = e_2 \ \equiv\ \nu X = f_2}{\nu X = e_1 \vee e_2 \ \equiv\ \nu X = f_1 \vee f_2}.$$

The notion of equivalence of Definition 7 is an equivalence relation on RESs and it satisfies the properties E1-E7 in Table 2. E1-E5 are proven for boolean equation systems in [14] and the proofs carry over to our setting. In the table, σ and σ' stand for the fixed-point symbols μ and ν. The equivalences E3 and E4 above give a method to solve arbitrary equation systems, provided a single equation can be solved. Here, solving a single equation $\sigma X{=}e$ means replacing it by an equivalent equation $\sigma X{=}e'$ where X does not occur in e', which is the topic of the next section. This method is known as Gauß-elimination as it resembles the well-known Gauß-elimination procedure for sets of linear equations [20].

The idea behind Gauß-elimination for a real equation system $\mathcal{E}$ is as follows. First, the last equation $\sigma_n X_n{=}e_n$ of $\mathcal{E}$ is solved for X_n. Assume the solution is $\sigma_n X_n{=}e'_n$, where X_n does not occur in e'_n. Using E3 the expression e'_n is substituted for all occurrences X_n in right-hand sides of $\mathcal{E}$ removing all occurrences of X_n except in the left hand side of the last equation. Subsequently, this process is repeated for the one but last equation of $\mathcal{E}$ up to the first equation. Now the first equation has the shape $X_1{=}e_1$ where no variable X_1 up till X_n occurs in e_1. Using E4 this equation can be moved to the end of $\mathcal{E}$, and by applying E3 all occurrences of X_1 are removed from the right-hand sides of $\mathcal{E}$. This is then repeated for X_2, which now also does not contain $X_1, \ldots, X_n$, until all variables $X_1, \ldots, X_n$ have been removed from all right-hand sides of $\mathcal{E}$.

A concrete, but simple example is the following. Consider the real equation system

$$\mu X{=}Y, \ \ \nu Y{=}(X+1) \wedge Y.$$

We can derive:

$$\mu X{=}Y, \ \nu Y{=}(X+1) \wedge Y \overset{(\dagger)}{\equiv} \mu X{=}Y, \ \nu Y{=}X+1 \overset{\text{E3}}{\equiv} \mu X{=}X+1, \ \nu Y{=}X+1 \overset{(\ddagger)}{\equiv}$$
$$\mu X{=}-\infty, \ \nu Y{=}X+1 \overset{\text{E4}}{\equiv} \nu Y{=}X+1, \ \mu X{=}-\infty, \overset{\text{E3}}{\equiv} \nu Y{=}-\infty, \ \mu X{=}-\infty.$$

Solving the equation $\nu Y = (X+1) \wedge Y$ at ($\dagger$) above, and $\mu X{=}X+1$ at ($\ddagger$) can be done with simple fixed-point iteration. In $\nu Y = (X+1) \wedge Y$ fixed-pointed iteration starts with $Y = \infty$. This yields in the first iteration $Y = X+1$, and this iteration is stable, and hence it is the maximal fixed-point solution. For $\mu X{=}X+1$, the initial approximation $X = -\infty$ is also a solution, and hence the minimal solution. Unfortunately, fixed-point iteration does not terminate always. For instance, $\mu X{=}(X+1) \vee 0$ has minimal solution $X = \infty$, which can only be obtained via an infinite number of iteration steps.

4 Solving single equations

In this section we show that it is possible to solve each fixed-point equation $\sigma X = e$ in a finite number of steps. First assume that e does not contain conditional operators. If we have a minimal fixed-point equation $\mu X{=}e$, we know via Theorem 4 that we can rewrite e to simple conjunctive normal form. We want to explicitly expose occurrences of the variable X in the normal form of e and do this by denoting the normal form of e as shown in (1). Here, all expressions containing variables different from X are moved to f_{ij} or m_i.

$$\bigwedge_{i\in I}\big(\bigvee_{j\in J_i}(c_{ij}{\cdot}X + c'_{ij}{\cdot}eq_{-\infty}(X) + f_{ij}) \vee m_i\big). \tag{1}$$

The expressions f_{ij} and m_i do not contain X. Subexpressions $c_{ij}{\cdot}X$ are optional, i.e., abusing notation, we allow c_{ij} to be 0 if this sub-term is not present. Likewise, $eq_{-\infty}(X)$ is optional and therefore, c'_{ij} is either 0 or 1, where 0 means that the expression is not present. Constants c_{ij} and c'_{ij} cannot both be 0, as in that case the conjunct does not contain X and is hence part of m_i.

We define the solution of $\mu X{=}e$, in which e is assumed to be of shape (1), as $\mu X = Sol^{\mu}_{X=e}$ where:

$$\begin{aligned} Sol^{\mu}_{X=e} = \bigwedge_{i\in I}\big((eq_{\infty}(\bigvee_{j\in J_i} f_{ij})) & \\ \Rightarrow (eq_{-\infty}(m_i) \Rightarrow -\infty \diamond ((\bigvee_{j\in J_i|c_{ij}\geq 1} f_{ij} + (c_{ij}-1){\cdot}U_i) \vee \bigvee_{j\in J_i|c'_{ij}=1} \infty \Rightarrow U_i \diamond \infty)) & \quad (2)\\ \diamond \quad \infty) & \end{aligned}$$

where $U_i = m_i \vee \bigvee_{j\in J_i|c_{ij}<1} \dfrac{1}{1-c_{ij}}{\cdot}f_{ij}$.

Note that we use the notation $\bigvee_{j\in J_i|cond}$ where $cond$ is a condition. This means that the disjunction is only taken over elements j that satisfy the condition. Also observe that we use expressions such as $\frac{1}{1-c_{ij}}{\cdot}f_{ij}$. This is an ordinary multiplication with $\frac{1}{1-c_{ij}}$ as positive constant. It is worth noting that if only rational numbers are used in the equations, the solutions to the variables are restricted to $-\infty$, ∞ and rationals.

It can be understood that (2) is a solution of (1) as follows. First observe that due to property E6 the solution of a minimal fixed-point distributes over the initial conjunction $\bigwedge_{i\in I}$ of clauses. This means that we can fix some $i \in I$ and only concentrate on understanding how one single clause $\bigvee_{j\in J_i}(c_{ij}{\cdot}X + c'_{ij}{\cdot}eq_{-\infty}(X) + f_{ij}) \vee m_i$ must be solved. If f_{ij} is equal to ∞ for some $j \in J_i$, the solution must be infinite. This is ensured by the outermost conditional operator in (2). Now, assuming that no f_{ij} is equal to ∞, we inspect m_i. If m_i equals $-\infty$, then the minimal solution for the given $i \in I$ is also $-\infty$. This explains the nested conditional operator in (2).

Next consider the innermost conditional operator of (2) and additionally assume $m_i > -\infty$. If there is some c'_{ij} that is equal to 1, then the minimal solution is at least m_i due to the disjunct m_i that appears in the clause. But then it must also be at least $1{\cdot}eq_{-\infty}(m_i) = \infty$. Hence, in this case the solution is ∞, which is ensured by the expression in the condition of the innermost conditional $\bigvee_{j\in J_i|c'_{ij}=1}\infty$. Otherwise, all c'_{ij} equal 0, and both the right-hand side of (1) and the solution (2) can be simplified to

$$\bigvee_{j\in J_i}(c_{ij}{\cdot}X + f_{ij}) \vee m_i \qquad \text{and} \qquad (\bigvee_{j\in J_i|c_{ij}\geq 1} f_{ij} + (c_{ij}-1){\cdot}U_i) \Rightarrow U_i \diamond \infty.$$

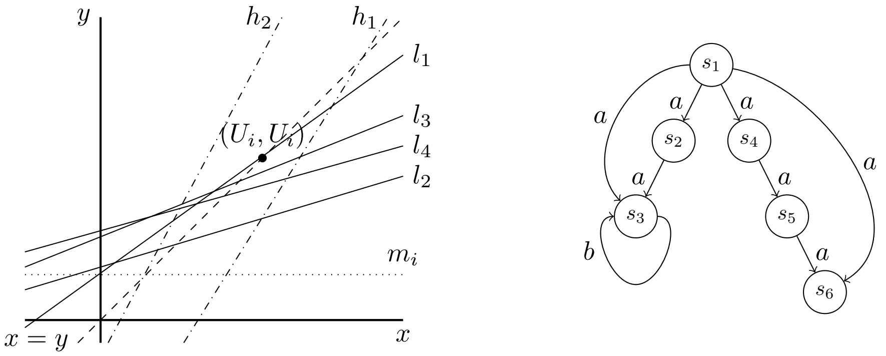


Figure 1 Solving a simple minimal fixed-point equation/An LTS with an infinite sequence of b's.

This resulting situation is best explained using Figure 1 (left). The simple conjunctive normal form consists of a number of disjunctions of the shape $c_{ij}\cdot X + f_{ij}$. These characterise lines of which we are interested in their intersection with the line $x = y$. In Figure 1 such lines are drawn as $l_1, \dots, l_4$, and h_1 and h_2. Due to the disjunction, we are interested in the maximal intersection point. If we first concentrate on those lines with $c_{ij} < 1$, then we see that (U_i, U_i) is the maximal intersection point of these lines above m_i. This intersection point is the solution for the equation unless there is a steep line, with $c_{ij} \geq 1$ which at $x = U_i$ lies above (U_i, U_i). In the figure there is such a line, *viz.* h_2. In such a case the fixed-point lies at the intersection of h_2 with the line $x = y$ for $x > U_i$. As this point does not exist in $\mathbb{R}$, the solution is ∞. The expression $\bigvee_{j\in J_i|c_{ij}\geq 1} f_{ij} + (c_{ij} - 1)\cdot U_i$ in (2) takes care of this situation. Steep lines, like h_1 which lie below (U_i, U_i) at $x = U_i$ can be ignored, as they do not force the minimal fixed-point U_i to become larger.

In case of a maximal fixed-point equation, $\nu X{=}e$ where e is a simple disjunctive normal form, it is useful to again expose the occurrences of X. We can denote the normal form of e in the following way:

$$\bigvee_{i\in I}\Big(\bigwedge_{j\in J_i}(c_{ij}\cdot X + c'_{ij}\cdot eq_{-\infty}(X) + f_{ij}) \wedge m_i\Big) \tag{3}$$

where $c_{ij}\cdot X$ and $eq_{-\infty}(X)$ are optional, i.e., c_{ij} can be 0, and c'_{ij} is either 0 or 1, where 0 means that the expression is not present. One of c_{ij} and c'_{ij} is not equal to 0. Again, the expressions f_{ij} and m_i do not contain X.

The solution of $\nu X{=}e$, where e is of the shape (3), is $\nu X = Sol^{\nu}_{X=e}$ with

$$\begin{aligned} Sol^{\nu}_{X=e} = \bigvee_{i\in I}\Big(&eq_{\infty}(m_i)\\ &\Rightarrow \Big(\bigwedge_{j\in J_i|c_{ij}\geq 1\wedge c'_{ij}=0}(f_{ij} + (c_{ij}-1))\cdot U_i\Big) \to -\infty \diamond U_i\\ &\diamond \quad \infty\Big) \end{aligned} \tag{4}$$

where $U_i = m_i \wedge \bigwedge_{j\in J_i|c_{ij}<1\wedge c'_{ij}=0} \dfrac{1}{1-c_{ij}}\cdot f_{ij}$.

The two fixed-point solutions are not syntactically dual which is due to the fact that simple conjunctive and disjunctive normal forms are not each other's dual, because of the presence of $+$ and $eq_{-\infty}$. We refrain from sketching the intuition underlying the solution to the maximal fixed-point as it is similar to that of the minimal fixed-point.

A full normal form can contain the conditional operators $e_1 \Rightarrow e_2 \diamond e_3$ and $e_1 \rightarrow e_2 \diamond e_3$. Suppose we have an equation $\sigma X = e_1 \Rightarrow e_2 \diamond e_3$ with σ either μ or ν. For the minimal fixed-point the right-hand side of the solution is $\mathit{Sol}^{\mu}_{X=e_1 \Rightarrow e_2 \diamond e_3} = (e_1[X := \mathit{Sol}^{\mu}_{X=e_2} \wedge \mathit{Sol}^{\mu}_{X=e_3}]) \Rightarrow \mathit{Sol}^{\mu}_{X=e_2} \diamond \mathit{Sol}^{\mu}_{X=e_3}$. For the maximal fixed-point we find the right-hand side $\mathit{Sol}^{\nu}_{X=e_1 \Rightarrow e_2 \diamond e_3} = (e_1[X := \mathit{Sol}^{\nu}_{X=e_3}]) \Rightarrow \mathit{Sol}^{\nu}_{X=e_2 \wedge e_3} \diamond \mathit{Sol}^{\nu}_{X=e_3}$.

In case of the other conditional operator $\sigma X = e_1 \rightarrow e_2 \diamond e_3$ we obtain for the right side of the minimal fixed-point $\mathit{Sol}^{\mu}_{X=e_1 \rightarrow e_2 \diamond e_3} = (e_1[X := \mathit{Sol}^{\mu}_{X=e_2}]) \rightarrow \mathit{Sol}^{\mu}_{X=e_2} \diamond \mathit{Sol}^{\mu}_{X=e_2 \vee e_3}$, and for the right side of the maximal fixed-point $\mathit{Sol}^{\nu}_{X=e_1 \rightarrow e_2 \diamond e_3} = (e_1[X := \mathit{Sol}^{\nu}_{X=e_2} \vee \mathit{Sol}^{\nu}_{X=e_3}]) \rightarrow \mathit{Sol}^{\nu}_{X=e_2} \diamond \mathit{Sol}^{\nu}_{X=e_3}$.

The following theorem summarises that these solutions solve fixed-point equations.

▶ **Theorem 11.** *For any fixed-point symbol σ, variable $X \in \mathcal{X}$ and expression e, it holds that*

$$\sigma X = e \;\equiv\; \sigma X = \mathit{Sol}^{\sigma}_{X=e}$$

and $X \notin \mathsf{occ}(\mathit{Sol}^{\sigma}_{X=e})$, where $\mathit{Sol}^{\sigma}_{X=e}$ is defined above.

Proof. By Theorem 4 we can assume that e is in normal form. The proof follows induction on the number of conditional operators. It is straightforward to see that, by construction, X does not occur in $\mathit{Sol}^{\sigma}_{X=e}$.

We only consider the case with a minimal fixed-point where e is a conjunctive normal form. Using property E6 it is possible to solve all conjuncts separately. So, without loss of generality, we assume that e has the shape

$$e = \bigvee_{j \in J} (c_j \cdot X + c'_j \cdot \mathit{eq}_{-\infty}(X) + f_j) \vee m \tag{5}$$

where $c_j \geq 0$ and $c'_j \in \{0,1\}$ are constants such that c_j and c'_j are not both 0, and f_j and m are expressions in which X does not occur. We show that the right-hand side of equation (2) without the initial conjunction provides the required term $\mathit{Sol}^{\mu}_{X=e}$ in this theorem. Concretely,

$$\begin{aligned} \mathit{Sol}^{\mu}_{X=e} = (&\mathit{eq}_{\infty}(\bigvee_{j \in J} f_j)) \\ &\Rightarrow (\mathit{eq}_{-\infty}(m) \Rightarrow -\infty \diamond (((\bigvee_{j \in J | c_j \geq 1} f_j + (c_j - 1) \cdot U) \vee \bigvee_{j \in J | c'_j = 1} \infty) \Rightarrow U \diamond \infty)) \\ &\diamond \quad \infty \end{aligned} \tag{6}$$

where $U = m \vee \bigvee_{j \in J | c_j < 1} \frac{1}{1 - c_j} \cdot f_j$.

Using Lemma 9 we must prove case 1 and 2 for a valuation η. We start with case 1. So, consider the smallest $r = \eta[X := r](e)$. We define $r' = \eta(\mathit{Sol}^{\mu}_{X=e})$ automatically satisfying the first proof obligation of Lemma 9, where it should be noted that X does not occur in $\mathit{Sol}^{\mu}_{X=e}$. Hence, we only need to show that $r' \leq r$. We distinguish a number of cases.

- Suppose there is some f_j such that $\eta[X := r](f_j) = \infty$. In that case both $r = \infty$ and $r' = \infty$. So, clearly, $r' \leq r$. Below we can now assume that there is no $j \in J$ such that $\eta[X := r](f_j) = \infty$.
- Now assume $\eta(m) = -\infty$. By the previous case we know that $f_j \neq \infty$. In that case $r' = \eta(\mathit{Sol}^{\mu}_{X=e}) = -\infty$, as $\eta(\mathit{eq}_{-\infty}(m)) = -\infty \leq 0$, and hence, $r' \leq r$. Below we assume that $\eta(m) \neq -\infty$.
- If there is at least one $j \in J$ such that $c'_j = 1$, then $r = \eta[X := r](e) = \infty$. The reason for this is that $r > -\infty$, as r at least has the value $\eta(m)$. But then $r = \infty$ as $\eta[X := r](c'_j \cdot \mathit{eq}_{-\infty}(X)) = \infty$. Clearly, $r' \leq r$. So, below we can assume that $c'_j = 0$ for all $j \in J$.

- With the assumptions above, we can write e more compactly.

  $$e = \bigvee_{j \in J} (c_j{\cdot}X + f_j) \vee m.$$

  We know that r is the smallest value satisfying

  $$r = \eta[X := r](e) = \eta[X := r](\bigvee_{j \in J} (c_j{\cdot}X + f_j) \vee m).$$

  Consider $r_1 = \eta(m \vee \bigvee_{j \in J | c_j < 1} (\frac{f_j}{1-c_j}))$.
  - First assume that there is no $j \in J$ with $c_j \geq 1$ such that $r_1 < \eta[X := r_1](c_j{\cdot}X + f_j)$. We show that r_1 is the solution, i.e., $r_1 = r$.
    Consider the case where $\eta(m) \geq \frac{\eta(f_j)}{1-c_j}$ for all $j \in J$ with $c_j < 1$. So, $r_1 = \eta(m)$. In this case $\eta(m)$ is a solution as (i) for those $j \in J$ for which $c_j < 1$, it holds that $\eta(m) \geq c_j{\cdot}\eta(m) + \eta(f_j)$, and (ii) by the assumption of this item for those $j \in J$ such that $c_j \geq 1$, also $\eta(m) < c_j{\cdot}\eta(m) + \eta(f_j)$. It is obvious that $\eta(m)$ must be the smallest solution.
    Now consider the case where $\eta(m) < \frac{\eta(f_j)}{1-c_j}$ for some $j \in J$. In this case $r_1 = \bigvee_{j \in J | c_j < 1} (\frac{\eta(f_j)}{1-c_j}) = \frac{\eta(f_{j'})}{1-c_{j'}}$ for some $j' \in J$, where j' is the index of the largest solution. It is straightforward to check that $\frac{\eta(f_{j'})}{1-c_{j'}}$ is a solution. It is also the smallest solution, which can be seen as follows. Suppose there were a smaller solution $r_2 < \frac{\eta(f_{j'})}{1-c_{j'}}$. Hence, $r_2 = \eta(m) \wedge \bigwedge_{j \in J} (c_j{\cdot}r_2 + \eta(f_j)) \geq c_{j'}{\cdot}r_2 + \eta(f_{j'})$. From this it follows that $r_2 \geq \frac{\eta(f_{j'})}{1-c_{j'}}$ contradicting that it is a smaller solution.
    It follows that $r_1 = r$ is the smallest solution. Furthermore, $r' = \eta(Sol^{\mu}_{X=e}) = \eta(U) = \eta(m \vee \bigvee_{j \in J | c_j < 1} \frac{f_j}{1-c_j}) = r_1 = r$. Obviously, $r' \leq r$.
  - Now assume that there is a $j \in J$ with $c_j \geq 1$ such that $r_1 < \eta[X := r_1](c_j{\cdot}X + f_j)$. We show that $r = \infty$. Using the argumentation of the previous item, the smallest solution r is at least r_1. But clearly, r_1 is larger than the non-infinite solution of $X = \eta[X := r_1](c_j{\cdot}X + f_j)$ as by the assumption $r_1 > \frac{\eta(f_j)}{1-c_j}$. Note that if $c_j > 1$, this solution exists, and if $c_j = 1$ there is only a finite solution if $f_j = 0$, but in this latter case the assumption of this item is invalid. Hence, the only remaining minimal solution is $r = \infty$. Clearly, for any choice of r' it holds that $r' < r$.

Now we concentrate on case 2 for the minimal fixed-point of Lemma 9. We know that $r = \eta(Sol^{\mu}_{X=e})$ is the minimal solution for $\eta(Sol^{\mu}_{X=e})$ and we must show that there is an $r' \leq r$ such that $r' \geq \eta[X := r'](e)$. We take $r' = r$ leaving us with the obligation to show that $r \geq \eta[X := r](e)$.

We distinguish the following cases.

- Assume that there is some f_j such that $\eta(f_j) = \infty$. In that case $r = \infty$, which satisfies $\infty \geq \eta[X := \infty](e)$. Below we assume that $\eta(f_j) < \infty$ for all $j \in J$.
- Now assume that $\eta(m) = -\infty$. Note that for any $j \in J$ it is the case that $c_j \neq 0$ or $c'_j \neq 0$. In this case, $r = -\infty$ is the solution as $\eta[X := -\infty](e) = -\infty$ and this implies our proof obligation. So, in the steps below we assume that $\eta(m) > -\infty$.
- With the conditions above, if there is at least one $j \in J$ such that $c'_j = 1$, then $r = \infty$ is the fixed-point satisfying our proof obligation. Below we assume that for all $j \in J$ it holds that $c'_j = 0$.
- As all c'_j can be assumed to be 0, we can simplify the equation for X to:

  $$\mu X = \bigvee_{j \in J} (c_j{\cdot}X + f_j) \vee m.$$

We find $\eta(U) = \eta(m \vee \bigvee_{j \in J | c_j < 1} \frac{f_j}{1-c_j})$. If there is no $j \in J$ with $c_j \geq 1$ such that $\eta(f_j - (1-c_j) \cdot U) > 0$ we find that $r = \eta(Sol^{\mu}_{X=e}) = \eta(U)$. We show that $r \geq \eta[X := r](e)$. If $\eta(m) \geq \bigvee_{j \in J | c_j < 1} \frac{\eta(f_j)}{1-c_j}$ then $r = \eta(m)$. For a $j \in J$ with $c_j < 1$ we find that $c_j \cdot \eta(m) + \eta(f_j) \leq \eta(m)$ as $\eta(m) \geq \frac{\eta(f_j)}{1-c_j}$. For a $j \in J$ with $c_j \geq 1$, we find by the condition above that $\eta(f_j + c_j \cdot U) \leq \eta(U)$, or in other words $\eta(f_j + c_j \cdot m) \leq \eta(m)$. So, $r = \eta(m) = \eta[X := r](e)$ as we had to show.

Otherwise, there is some $j' \in J$ with $c_{j'} < 1$ such that $\frac{\eta(f_{j'})}{1-c_{j'}} = \bigvee_{j \in J | c_j < 1} \frac{\eta(f_j)}{1-c_j}$. In this case $r = \frac{\eta(f_{j'})}{1-c_{j'}}$. From the conditions, we can see that $r = \eta[X := r](e)$ as we had to show.

- Now assume that there is a $j \in J$ with $c_j \geq 1$ such that $\eta(f_j - (1 - c_j) \cdot U) > 0$. In this case $r = \eta(Sol^{\mu}_{X=e}) = \infty$, clearly satisfying our proof obligation.

This finishes our proof for a minimal fixed-point equation. ◀

5 Relation to boolean equation systems

A boolean equation system (BES) is a restricted form of a real equation system where solutions can only be *true* or *false* [20]. Concretely, the syntax for expressions is

$$e \ ::= \ X \mid \mathit{true} \mid \mathit{false} \mid e \vee e \mid e \wedge e$$

where X is taken from some set $\mathcal{X}$ of variables [20]. A boolean equation system is a sequence of fixed-point equations $\sigma_1 X_1 = e_1, \ldots, \sigma_n X_n = e_n$ where σ_i are fixed-point operators, X_i are variables from $\mathcal{X}$ ranging over *true* and *false*, and e_i are boolean expressions.

We do not spell out the semantics of boolean equation systems, as it is similar to that of RESs. However, we believe that it is useful to indicate the relation with real equation systems.

The simplest embedding is where a given BES is literally transformed to a RES and *true* and *false* are interpreted as ∞ and $-\infty$. We consider a minimal fixed-point equation. The right-hand side can be rewritten to a simple conjunctive normal form. We write this in the shape of equation (1). So, $c_{ij} = 1$, $c'_{ij} = 0$, f_{ij} is absent and m_i does not contain X and can only be interpreted as $\pm\infty$. Exactly if J_i is not empty, X is present in conjunct i.

$$\mu X = \bigwedge_{i \in I} ((\bigvee_{j \in J_i} X) \vee m_i).$$

The solution is given by equation (2), which can be simplified to:

$$\bigwedge_{i \in I} (eq_{-\infty}(m_i) \Rightarrow -\infty \diamond ((\bigvee_{j \in J_i} 0) \Rightarrow m_i \diamond \infty)) = \bigwedge_{i \in I} m_i = \bigwedge_{i \in I} ((\bigvee_{j \in J_i} -\infty) \vee m_i).$$

The latter exactly coincides with the Gauß-elimination rule for BESs that says that in an equation $\mu X = e$, any occurrence of X in e can safely be replaced by *false*. For the maximal fixed-point operator, dual reasoning applies. As Gauß-elimination is a complete way to solve a BES with *true* and *false*, and exactly the same reduction works with the corresponding RES with ∞ and $-\infty$, this confirms that this interpretation works.

An alternative interpretation is given by taking two arbitrary constants c_{true} and c_{false} with as only constraint that $c_{\mathit{true}} > c_{\mathit{false}}$. A boolean equation system $\sigma_1 X_1 = e_1, \ldots, \sigma_n X_n = e_n$ is translated into $\sigma_1 X_1 = c_{\mathit{false}} \vee (c_{\mathit{true}} \wedge e_1), \ldots, \sigma_n X_n = c_{\mathit{false}} \vee (c_{\mathit{true}} \wedge e_n)$ of which the validity can be established in the same way as above.

6 Quantitative modal formulas and their translation to RESs

We can write quantitative modal formulas that yield a value instead of true and false. In the next section we provide examples of what can be expressed. Our formulas have the syntax

$$\phi \ ::= \ X \mid d \mid c{\cdot}\phi \mid \phi + \phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \langle a \rangle \phi \mid [a]\phi \mid \mu X.\phi \mid \nu X.\phi.$$

Here $d \in \hat{\mathbb{R}}$ and $c \in \mathbb{R}$ with $c > 0$ are constants, $X \in \mathcal{X}$ is a variable, and $a \in \mathcal{A}$ is an action from some set of actions $\mathcal{A}$. Although there are many similar logics around, we have not encountered this exact form before.

We evaluate these modal formulas on probabilistic LTSs. For a finite set of states S, we use distributions $d : S \to [0,1]$ where $d(s)$ is the probability to end up in state s. Distributions satisfy that $\sum_{s \in S} d(s) = 1$. The set of all distributions over S is denoted by $\mathcal{D}(S)$.

▶ **Definition 12.** *A probabilistic labelled transition system (pLTS) is a four-tuple $M = (S, \mathcal{A}, \to, d_0)$ where S is a finite set of states, $\mathcal{A}$ is a finite set of actions, the relation $\to \subseteq S \times \mathcal{A} \times \mathcal{D}(S)$ represents the transition relation, and $d_0 \in \mathcal{D}(S)$ is the initial distribution.*

We leave out the definition of the interpretation of quantitative modal formulas on probabilistic LTSs, as it is standard. Instead, we define the real equation system that is generated given a modal formula ϕ and a probabilistic labelled transition system $M = (S, \mathcal{A}, \to, d_0)$, following the translations in [20, 14, 21, 11]. The function $Eq(\phi)$ generates the required sequence of RES equations for ϕ and $rhs(s, \phi)$ yields the expression for the right-hand side of such an equation representing the value of ϕ in state s.

$$\begin{array}{ll}
Eq(X) = \epsilon, & rhs(s, X) = X_s, \\
Eq(d) = \epsilon, & rhs(s, d) = d, \\
Eq(c{\cdot}\phi) = Eq(\phi), & rhs(s, c{\cdot}\phi) = c{\cdot}rhs(s, \phi), \\
Eq(\phi_1 + \phi_2) = Eq(\phi_1), Eq(\phi_2), & rhs(s, \phi_1 + \phi_2) = rhs(s, \phi_1) + rhs(s, \phi_2), \\
Eq(\phi_1 \vee \phi_2) = Eq(\phi_1), Eq(\phi_2), & rhs(s, \phi_1 \vee \phi_2) = rhs(s, \phi_1) \vee rhs(s, \phi_2), \\
Eq(\phi_1 \wedge \phi_2) = Eq(\phi_1), Eq(\phi_2), & rhs(s, \phi_1 \wedge \phi_2) = rhs(s, \phi_1) \wedge rhs(s, \phi_2), \\
Eq(\langle a \rangle \phi) = Eq(\phi), & rhs(s, \langle a \rangle \phi) = \bigvee_{\{d \in \mathcal{D}(S) \mid s \xrightarrow{a} d\}} \sum_{s' \in S} d(s'){\cdot}rhs(s', \phi), \\
Eq([a]\phi) = Eq(\phi), & rhs(s, [a]\phi) = \bigwedge_{\{d \in \mathcal{D}(S) \mid s \xrightarrow{a} d\}} \sum_{s' \in S} d(s'){\cdot}rhs(s', \phi), \\
Eq(\mu X.\phi) = \langle \mu X_s = rhs(s, \phi) \mid s \in S \rangle, Eq(\phi), & rhs(s, \mu X.\phi) = X_s, \\
Eq(\nu X.\phi) = \langle \nu X_s = rhs(s, \phi) \mid s \in S \rangle, Eq(\phi). & rhs(s, \nu X.\phi) = X_s.
\end{array}$$

We use the notation $\langle \sigma X_s = e_s \mid s \in S \rangle$ for the sequence of all equations $\sigma X_s = e_s$ for all states $s \in S$.

The evaluation of a modal formula ϕ in M with initial distribution d_0 is the solution in $\hat{\mathbb{R}}$ of variable X_{init} in the RES $\mu X_{init} = (\sum_{s \in S} d_0(s){\cdot}rhs(s, \phi))$, $Eq(\phi)$. The use of the minimal fixed-point for the initial variable is of no consequence as X_{init} does not occur elsewhere in the equation system. A maximal fixed-point could also be used.

7 Applications

7.1 The longest a-sequence to a b-loop

We are interested in the longest sequence of actions a to reach a state where an infinite sequence of actions b can be done. The modal formula that expresses this is the following:

$$\mu X.(1 + \langle a \rangle X) \vee (0 \wedge \nu Y.\langle b \rangle Y).$$

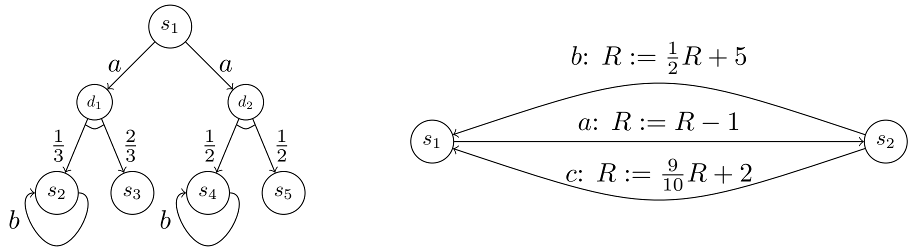


Figure 2 A probabilistic LTS with a loop/An LTS with rewards.

The last part with the maximal fixed-point $0 \wedge \nu Y.\langle b\rangle Y$ when evaluated in a state equals $-\infty$ if no infinite sequence of b's is possible. Otherwise, it evaluates to 0. The first part $1 + \langle a\rangle X$ yields 1 plus the maximum values of the evaluation of X in all states reachable by an action a. If no infinite b-sequence can be reached from such a state, this value is $-\infty$, and otherwise it represents the maximal number of steps to reach such an infinite b-sequence.

We evaluate this formula in the labelled transition system given at the right in Figure 1. This leads to the following real equation system where X_i and Y_i correspond to the value of X, resp. Y in state s_i. The solution of the equation system is written behind each equation.

$$\begin{array}{lr@{\qquad}lr}
\mu X_1 = (1 + (X_2 \vee X_3 \vee X_4 \vee X_6)) \vee (0 \wedge Y_1) & 2 & \nu Y_1 = -\infty & -\infty \\
\mu X_2 = (1 + X_3) \vee (0 \wedge Y_2) & 1 & \nu Y_2 = -\infty & -\infty \\
\mu X_3 = (1 + -\infty) \vee (0 \wedge Y_3) & 0 & \nu Y_3 = Y_3 & \infty \\
\mu X_4 = (1 + X_5) \vee (0 \wedge Y_4) & -\infty & \nu Y_4 = -\infty & -\infty \\
\mu X_5 = (1 + X_6) \vee (0 \wedge Y_5) & -\infty & \nu Y_5 = -\infty & -\infty \\
\mu X_6 = (1 + -\infty) \vee (0 \wedge Y_6) & -\infty & \nu Y_6 = -\infty & -\infty
\end{array}$$

We find that the longest sequence of actions a is 2, which matches our expectation.

7.2 The probability to reach a loop

We are interested in the probability to reach a b-loop. We apply it to the LTS at the left in Figure 2. Due to the non-determinism there are more paths to such loops, and we are interested in the path with the highest probability. This is expressed by the modal formula

$$\mu X.\langle a\rangle X \vee \langle b\rangle X \vee ((\nu Y.\langle b\rangle Y \vee 0) \wedge 1).$$

As we want a probability, we use $\_ \wedge 1$ and $\_ \vee 0$ to enforce that the solution is in $[0, 1]$. The formula $\nu Y.\langle b\rangle Y \vee 0$ yields ∞ if an infinite sequence of actions b is possible and 0 otherwise.

The translation of this formula on the labelled transition system in Figure 2 yields the following real equation system.

$$\begin{array}{lcl@{\qquad}lcl}
\mu X_1 = (\frac{1}{3}{\cdot}X_2 + \frac{2}{3}{\cdot}X_3) \vee (\frac{1}{2}{\cdot}X_4 + \frac{1}{2}{\cdot}X_5) \vee (Y_1 \wedge 1) & & & \nu Y_1 = -\infty \vee 0 & = & 0, \\
 & = & \frac{1}{3} \vee \frac{1}{2} \vee 0 = \frac{1}{2}, & & & \\
\mu X_2 = X_2 \vee (Y_2 \wedge 1) & = & X_2 \vee 1 = 1, & \nu Y_2 = Y_2 & = & \infty, \\
\mu X_3 = -\infty \vee (Y_3 \wedge 1) & = & -\infty \vee 0 = 0, & \nu Y_3 = -\infty \vee 0 & = & 0, \\
\mu X_4 = X_4 \vee (Y_4 \wedge 1) & = & X_4 \vee 1 = 1, & \nu Y_4 = Y_4 & = & \infty, \\
\mu X_5 = -\infty \vee (Y_5 \wedge 1) & = & -\infty \vee 0 = 0, & \nu Y_5 = -\infty \vee 0 & = & 0.
\end{array}$$

This shows that the maximal probability to reach a b-loop is $\frac{1}{2}$.

7.3 Determining the reward of process behaviour

In Figure 2 at the right a labelled transition system is drawn, where a reward R is changed when a transition takes place. The transition labelled with action a costs one unit, b yields $\frac{1}{2}R + 5$ units, and the transition c adapts the reward by $\frac{9}{10}R + 2$. We want to know what the maximal stable reward is. This is expressed by the following formula:

$$\mu R.\langle a\rangle(R-1) \vee \langle b\rangle(\tfrac{1}{2}{\cdot}R+5) \vee \langle c\rangle(\tfrac{9}{10}{\cdot}R+2) \vee 0.$$

Note that we express this as the minimal reward larger than 0, which is the maximum of all individual rewards. Translating this to a real equation system yields

$$\mu R_1 = (R_2-1) \vee -\infty \vee -\infty \vee 0, \qquad \mu R_2 = -\infty \vee (\tfrac{1}{2}{\cdot}R_1+5) \vee (\tfrac{9}{10}R_1+2) \vee 0.$$

We solve this using Gauß-elimination. This means that the second equation is substituted in the first, which, after some straightforward simplifications, gives us

$$\mu R_1 = (\tfrac{1}{2}{\cdot}R_1+4) \vee (\tfrac{9}{10}{\cdot}R_1+1) \vee 0.$$

We solve this equation using the technique of Section 4, leading to:

$$R_1 = \frac{4}{1-\frac{1}{2}} \vee \frac{1}{1-\frac{9}{10}} \vee 0 = 10.$$

8 Conclusions and outlook

We introduce real equation systems (RESs) as the pendant of Boolean Equation Systems with solutions in the domain of the reals extended with ±∞. By a number of examples we show how this can be used to evaluate a wide range of quantitative properties of process behaviour.

We provide a complete method to solve RESs using an extension of what is called "Gauß-elimination" [21] to solve boolean equation systems. It shows that any RES can be solved by carrying out a finite number of substitutions. As solving RESs generalises solving BESs, and Gauß-elimination on BESs is exponential, our Gauß-elimination technique can also lead to exponential growth of intermediate terms. A prototype implementation shows that depending on the nature of the system being analysed, this may or may not be an issue. For instance, analysing the Game of the Goose [12] or The Ant on a Grid [6], are practically undoable with the method proposed here, while the Lost Boarding Pass Problem [10] is easily solved, even for planes with 100,000 passengers.

We believe that the next step is to come up with algorithms that are more efficient in practice than Gauß-elimination. This is motivated by the situation with BESs where for instance the recursive algorithm [23, 26] turns out to be practically far more efficient than Gauß-elimination [9].

References

1 Giorgio Bacci, Giovanni Bacci, Mathias Claus Jensen, and Kim G. Larsen. Convex lattice equation systems. In Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Rupak Majumdar, editors, *Principles of Systems Design – Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, volume 13660 of *Lecture Notes in Computer Science*, pages 438–455. Springer, 2022. doi:10.1007/978-3-031-22337-2_21.

2 Julian C. Bradfield and Colin Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3 of *Studies in logic and practical reasoning*, pages 721–756. North-Holland, 2007.

3 Julian C. Bradfield and Igor Walukiewicz. The mu-calculus and model checking. In *Handbook of Model Checking*, pages 871–919. Springer, 2018.

4 Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 252–263. ACM, 2017. doi:10.1145/3055399.3055409.

5 Sjoerd Cranen, Jan Friso Groote, and Michel A. Reniers. A linear translation from CTL* to the first-order modal μ-calculus. *Theor. Comput. Sci.*, 412(28):3129–3139, 2011. doi:10.1016/j.tcs.2011.02.034.

6 Susmoy Das and Arpit Sharma. On the use of model and logical embeddings for model checking of probabilistic systems. In Marieke Huisman and António Ravara, editors, *Formal Techniques for Distributed Objects, Components, and Systems – 43rd IFIP WG 6.1 International Conference, FORTE 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, volume 13910 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2023. doi:10.1007/978-3-031-35355-0_8.

7 Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 – April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2007. doi:10.1007/978-3-540-71316-6_21.

8 Thomas Martin Gawlitza and Helmut Seidl. Solving systems of rational equations through strategy iteration. *ACM Trans. Program. Lang. Syst.*, 33(3):11:1–11:48, 2011. doi:10.1145/1961204.1961207.

9 Maciej Gazda and Tim A. C. Willemse. Zielonka's recursive algorithm: dull, weak and solitaire games and tighter bounds. In Gabriele Puppis and Tiziano Villa, editors, *Proceedings Fourth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2013, Borca di Cadore, Dolomites, Italy, 29-31th August 2013*, volume 119 of *EPTCS*, pages 7–20, 2013. doi:10.4204/EPTCS.119.4.

10 Jan Friso Groote and Erik P. de Vink. Problem solving using process algebra considered insightful. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2017. doi:10.1007/978-3-319-68270-9_3.

11 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems.* MIT Press, 2014. URL: https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems.

12 Jan Friso Groote, Freek Wiedijk, and Hans Zantema. A probabilistic analysis of the game of the goose. *SIAM Rev.*, 58(1):143–155, 2016. doi:10.1137/140983781.

13 Jan Friso Groote and Tim A. C. Willemse. Model-checking processes with data. *Sci. Comput. Program.*, 56(3):251–273, 2005. doi:10.1016/j.scico.2004.08.002.

14 Jan Friso Groote and Tim A. C. Willemse. Parameterised boolean equation systems. *Theor. Comput. Sci.*, 343(3):332–369, 2005. doi:10.1016/j.tcs.2005.06.016.

15 Thomas A. Henzinger. From boolean to quantitative notions of correctness. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 157–158. ACM, 2010. doi:10.1145/1706299.1706319.

16 Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. doi:10.1007/11813040_1.

17 Marcin Jurdzinski and Ranko Lazic. Succinct progress measures for solving parity games. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–9. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005092.

18 Kyriakos Kalorkoti. Solving Łiukasiewicz μ-terms. *Theor. Comput. Sci.*, 712:38–49, 2018. doi:10.1016/j.tcs.2017.11.002.

19 Kim Guldstrand Larsen. Efficient local correctness checking. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 – July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 1992. doi:10.1007/3-540-56496-9_4.

20 Angelika Mader. Modal μ-calculus, model checking and gauß elimination. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings*, volume 1019 of *Lecture Notes in Computer Science*, pages 72–88. Springer, 1995. doi:10.1007/3-540-60630-0_4.

21 Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems.* PhD thesis, Technische Universität München, 1997.

22 Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.

23 Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl. Logic*, 65(2):149–184, 1993. doi:10.1016/0168-0072(93)90036-D.

24 Matteo Mio and Alex Simpson. Łukasiewicz μ-calculus. *Fundam. Informaticae*, 150(3-4):317–346, 2017. doi:10.3233/FI-2017-1472.

25 Tom van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 291–308. Springer, 2018. doi:10.1007/978-3-319-89960-2_16.

26 Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998. doi:10.1016/S0304-3975(98)00009-7.

Constraint Automata on Infinite Data Trees: from CTL($\mathbb{Z}$)/CTL*($\mathbb{Z}$) to Decision Procedures

Stéphane Demri
Université Paris-Saclay, LMF, CNRS, ENS Paris-Saclay, France

Karin Quaas[1]
Fakultät für Mathematik und Informatik, Universität Leipzig, Germany

Abstract

We introduce the class of tree constraint automata with data values in $\mathbb{Z}$ (equipped with the less than relation and equality predicates to constants), and we show that the nonemptiness problem is ExpTime-complete. Using an automata-based approach, we establish that the satisfiability problem for CTL($\mathbb{Z}$) (CTL with constraints in $\mathbb{Z}$) is ExpTime-complete, and the satisfiability problem for CTL*($\mathbb{Z}$) is 2ExpTime-complete (only decidability was known so far). By-product results with other concrete domains and other logics, are also briefly discussed.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Constraints, Constraint Automata, Temporal Logics, Infinite Data Trees

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.29

Related Version *Full Version*: https://arxiv.org/pdf/2302.05327.pdf [27]

1 Introduction

In this paper, we study the satisfiability problem for the branching-time temporal logics CTL($\mathbb{Z}$) and CTL*($\mathbb{Z}$), extending the classical temporal logics CTL and CTL* in that atomic formulae express constraints about the relational structure $(\mathbb{Z}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{Z}})$. Formulae in these logics are interpreted over Kripke structures that are annotated with values in $\mathbb{Z}$. A typical CTL*($\mathbb{Z}$) formula is the formula $\mathsf{AGF}(\mathtt{x} < \mathsf{X}\mathtt{x})$ stating that on all paths infinitely often the value of the variable $\mathtt{x}$ at the current position is strictly smaller than the value of $\mathtt{x}$ at the next position. Formalisms defined over relational structures, also known as *concrete domains*, are considered in many works, including works on temporal logics [40, 13, 54, 48, 19, 35], description logics [50, 51, 52, 53, 16, 45, 3], and automata [38, 61, 43, 71, 65, 57]. Combining reasoning in your favourite logic with reasoning in a relevant concrete domain reveals to be essential for numerous applications, for instance for reasoning about ontologies, see e.g. [52, 46], or data-aware systems, see e.g. [28, 34]. A brief survey can be found in [26].

Decidability results for concrete domains handled in [53, 38, 3] exclude the ubiquitous concrete domain $(\mathbb{Z}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{Z}})$. By contrast, decidability results for logics with concrete domain $\mathbb{Z}$ require dedicated proof techniques, see e.g. [11, 23, 61, 46]. In particular, *fragments* of CTL*($\mathbb{Z}$) are shown decidable in [11] using integral relational automata from [17], and the satisfiability problem for existential and universal CTL* with gap-order constraints (more general than the ones in this paper) can be solved in PSpace [12, Theorem 14]. Another important breakthrough came with the decidability of CTL*($\mathbb{Z}$) [15, Theorem 32] (see also [14]) by designing a reduction to a decidable second-order logic, whose formulae are made of Boolean combinations of formulae from MSO and from WMSO+U [10], where U is the unbounding second-order quantifier, see e.g. [8, 9]. This is all the more remarkable as the decidability result is part of a powerful general approach [15], but no sharp complexity

[1] Supported by the Deutsche Forschungsgemeinschaft (DFG), project 504343613.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 29; pp. 29:1–29:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

upper bound can be inferred. More recently, the condition $C_{\mathbb{Z}}$ [23] to approximate the set of satisfiable symbolic models of a given LTL($\mathbb{Z}$) formula is extended to the branching case in [46] leading to the ExpTime-easiness of a major reasoning task for the description logic $\mathcal{ALCF}^{\mathcal{P}}(\mathbb{Z}_c)$. However, no elementary complexity upper bounds for the satisfiability problem for CTL($\mathbb{Z}$) nor CTL*($\mathbb{Z}$) were known since their decidability was established in [13, 15].

In this paper, we prove that the satisfiability problem for CTL($\mathbb{Z}$) is ExpTime-complete, and the satisfiability problem for CTL*($\mathbb{Z}$) is 2ExpTime-complete. We pursue the *automata-based approach* for solving decision problems for temporal logics, following seminal works for temporal logics, see e.g. [68, 69, 44]. This popular approach consists of reducing logical problems (satisfiability, model-checking) to automata-based decision problems while taking advantage of existing results and decision procedures from automata theory, see e.g. [67].

It is well-known that decision procedures for CTL* are difficult to design, and the combination with the concrete domain $\mathbb{Z}$ is definitely challenging. Moreover, we aim at proposing a general framework: we do not wish for every new logic with concrete domain to study again and again what is the proper way to define products of automata leading to optimal complexity. That is why our main goal in this work is to investigate a new class of *tree constraint automata*, understood as a target formalism in the pure tradition of the automata-based approach, and easy to reuse. The structures accepted by such tree constraint automata are *infinite* trees in which nodes are labelled by a letter from a finite alphabet and a tuple in $\mathbb{Z}^{\beta}$ for some $\beta \geqslant 1$ (this excludes the automata designed in [36, 37] dedicated to finite trees where no predicate $<$ is involved). Decision problems for alternating automata over infinite alphabets are often undecidable, see e.g. [56, 47, 25, 41], and therefore we advocate the introduction of *nondeterministic* constraint automata without alternation. Our definition of tree constraint automata naturally extends the definition of constraint automata for words (see e.g. [17, 59, 61, 43, 57]) and as far as we know, the extension to infinite trees in the way done herein has not been considered earlier in the literature.

As a key result, we show that the nonemptiness problem for tree constraint automata over $(\mathbb{Z}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{Z}})$ is ExpTime-complete. In order to obtain the ExpTime upper bound, we adapt results from [46, 45] (originally expressed in the context of interpretations for description logics) and we take advantage of several automata-based constructions for Rabin/Streett tree automata. As a corollary, we establish that the satisfiability problem for CTL($\mathbb{Z}$) is ExpTime-complete (Theorem 14), which is one of the main results of the paper. As a by-product, it also allows us to conclude that the concept satisfiability problem w.r.t. general TBoxes for $\mathcal{ALCF}^{\mathcal{P}}(\mathbb{Z}_c)$ is in ExpTime, a result known since [46].

Our main contribution is the characterisation of the complexity for CTL*($\mathbb{Z}$) satisfiability, which is an open problem evoked in [15, Section 9] and [46, Section 5] (decidability was established ten years ago in [14]). In Section 6, we show that the satisfiability problem for CTL*($\mathbb{Z}$) is in 2ExpTime by using Rabin tree constraint automata (introduced herein). We have to check that the essential steps for CTL* can be lifted to CTL*($\mathbb{Z}$) to get the optimal upper bound. In general, our contributions stem from the cross-fertilisation of automata-based techniques for temporal logics and reasoning about (infinite) structures made of $\mathbb{Z}$-constraints.

A complete version with all the proofs can be found in [27].

2 Temporal Logics with Numerical Domains

2.1 Concrete Domain $(\mathbb{Z}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{Z}})$ and Kripke Structures

In the sequel, we consider the concrete domain $(\mathbb{Z}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{Z}})$ (also written $\mathbb{Z}$), where $=_{\mathfrak{d}}$ is a unary predicate stating the equality with the constant $\mathfrak{d}$ and, $<$ and $=$ are the usual relations on $\mathbb{Z}$. Let $\mathrm{VAR} = \{\mathtt{x}, \mathtt{y}, \ldots\}$ be a countably infinite set of variables. A *term* $\mathtt{t}$ *over* VAR is an expression of the form $\mathsf{X}^i\mathtt{x}$, where $\mathtt{x} \in \mathrm{VAR}$ and X^i is a (possibly empty) sequence of i symbols "X". A term $\mathsf{X}^i\mathtt{x}$ should be understood as a variable (that needs to be interpreted) but, later on, we will see that the prefix X^i will have a temporal interpretation. We write $\mathrm{T}_{\mathrm{VAR}}$ to denote the set of all terms over VAR. For all $i \in \mathbb{N}$, we write $\mathrm{T}_{\mathrm{VAR}}^{\leqslant i}$ to denote the subset of terms of the form $\mathsf{X}^j\mathtt{x}$, where $j \leqslant i$. For instance, $\mathrm{T}_{\mathrm{VAR}}^{\leqslant 0} = \mathrm{VAR}$. An *atomic constraint* θ *over* $\mathrm{T}_{\mathrm{VAR}}$ is an expression of one of the forms below:

$$\mathtt{t} < \mathtt{t}' \qquad \mathtt{t} = \mathtt{t}' \qquad =_{\mathfrak{d}} (\mathtt{t}) \text{ (also written } \mathtt{t} = \mathfrak{d}),$$

where $\mathfrak{d} \in \mathbb{Z}$ and $\mathtt{t}, \mathtt{t}' \in \mathrm{T}_{\mathrm{VAR}}$. A *constraint* Θ is defined as a Boolean combination of atomic constraints. Constraints are interpreted on valuations $\mathfrak{v} : \mathrm{T}_{\mathrm{VAR}} \to \mathbb{Z}$ that assign elements from $\mathbb{Z}$ to the terms in $\mathrm{T}_{\mathrm{VAR}}$, so that $\mathfrak{v}$ *satisfies* θ, written $\mathfrak{v} \models \theta$, if and only if, the interpretation of the terms in θ makes θ true in $\mathbb{Z}$ in the usual way. The Boolean connectives are interpreted as usual. A constraint Θ is *satisfiable* $\overset{\text{def}}{\Leftrightarrow}$ there is a valuation $\mathfrak{v} : \mathrm{T}_{\mathrm{VAR}} \to \mathbb{Z}$ such that $\mathfrak{v} \models \Theta$. Similarly, a constraint Θ_1 *entails* a constraint Θ_2 (written $\Theta_1 \models \Theta_2$) $\overset{\text{def}}{\Leftrightarrow}$ for all valuations $\mathfrak{v}$, we have $\mathfrak{v} \models \Theta_1$ implies $\mathfrak{v} \models \Theta_2$. The satisfiability problem restricted to finite conjunctions of atomic constraints can be solved in PTime (see e.g. [17, Lemma 5.5]) and entailment is in coNP. In the sequel, quite often, the valuations $\mathfrak{v}$ are of the form $\{\mathtt{x}_1, \ldots, \mathtt{x}_\beta\} \to \mathbb{Z}$ when we are only interested in the values for the variables in $\{\mathtt{x}_1, \ldots, \mathtt{x}_\beta\}$.

Kripke structures. In order to define logics with the concrete domain $\mathbb{Z}$, the semantical structures of such logics are enriched with valuations that interpret the variables by elements in $\mathbb{Z}$. A *$\mathbb{Z}$-decorated Kripke structure* (or *Kripke structure* for short) $\mathcal{K}$ is a triple $(\mathcal{W}, \mathcal{R}, \mathfrak{v})$, where $\mathcal{W}$ is a non-empty set of *worlds*, $\mathcal{R} \subseteq \mathcal{W} \times \mathcal{W}$ is the accessibility relation and $\mathfrak{v} : \mathcal{W} \times \mathrm{VAR} \to \mathbb{Z}$ is a valuation function. A Kripke structure $\mathcal{K}$ is *total* whenever for all $w \in \mathcal{W}$, there is $w' \in \mathcal{W}$ such that $(w, w') \in \mathcal{R}$. Given a Kripke structure $\mathcal{K} = (\mathcal{W}, \mathcal{R}, \mathfrak{v})$ and a world $w \in \mathcal{W}$, an *infinite path* π from w is an ω-sequence $w_0, w_1 \ldots w_n, \ldots$ such that $w_0 = w$ and for all $i \in \mathbb{N}$, we have $(w_i, w_{i+1}) \in \mathcal{R}$. Finite paths are defined accordingly.

Labelled trees. Given $D \geqslant 1$, a *labelled tree of degree* D is a map $\mathbb{t} : dom(\mathbb{t}) \to \Sigma$ where Σ is some (potentially infinite) alphabet and $dom(\mathbb{t})$ is an infinite subset of $[0, D-1]^*$ such that $\mathbf{n} \in dom(\mathbb{t})$ and $\mathbf{n} \cdot i \in dom(\mathbb{t})$ for all $0 \leqslant i < j$ whenever $\mathbf{n} \cdot j \in dom(\mathbb{t})$ for some $\mathbf{n} \in [0, D-1]^*$ and $j \in [0, D-1]$. The elements of $dom(\mathbb{t})$ are called *nodes*. The empty word ε is the *root node* of $\mathbb{t}$. For every $\mathbf{n} \in dom(\mathbb{t})$, the elements $\mathbf{n} \cdot i$ ($i \in [0, D-1]$) are called the *children nodes of* $\mathbf{n}$, and $\mathbf{n}$ is called the *parent node of* $\mathbf{n} \cdot i$. We say that the tree $\mathbb{t}$ is a *full D-ary tree* if every node $\mathbf{n}$ has exactly D children $\mathbf{n} \cdot 0, \ldots, \mathbf{n} \cdot (D-1)$. Given a tree $\mathbb{t}$ and a node $\mathbf{n}$ in $dom(\mathbb{t})$, an infinite *path* in $\mathbb{t}$ starting from $\mathbf{n}$ is an infinite sequence $\mathbf{n} \cdot j_1 \cdot j_2 \cdot j_3 \ldots$, where $j_i \in [0, D-1]$ and $\mathbf{n} \cdot j_1 \ldots j_i \in dom(\mathbb{t})$ for all $i \geqslant 1$.

A *tree Kripke structure* $\mathcal{K}$ is a Kripke structure $(\mathcal{W}, \mathcal{R}, \mathfrak{v})$ such that $(\mathcal{W}, \mathcal{R})$ is a tree (not necessarily a full D-ary tree). Tree Kripke structures $(\mathcal{W}, \mathcal{R}, \mathfrak{v})$ such that $(\mathcal{W}, \mathcal{R})$ is isomorphic to the tree induced by $[0, D-1]^*$ are represented by maps of the form $\mathbb{t} : [0, D-1]^* \to \mathbb{Z}^\beta$. This assumes that we only care about the value of the variables $\mathtt{x}_1, \ldots, \mathtt{x}_\beta$ and $\mathbb{t}(\mathbf{n}) = (\mathfrak{d}_1, \ldots, \mathfrak{d}_\beta)$ encodes that for all $i \in [1, \beta]$, we have $\mathfrak{v}(\mathbf{n}, \mathtt{x}_i) = \mathfrak{d}_i$.

2.2 The Logic CTL*$(\mathbb{Z})$

We introduce the logic CTL*$(\mathbb{Z})$ extending the temporal logic CTL* from [29] but with constraints over $\mathbb{Z}$. *State formulae* ϕ and *path formulae* Φ of CTL*$(\mathbb{Z})$ are defined below

$$\phi := \neg\phi \mid \phi \wedge \phi \mid \mathsf{E}\Phi \qquad \Phi := \phi \mid \mathtt{t} = \mathfrak{d} \mid \mathtt{t}_1 = \mathtt{t}_2 \mid \mathtt{t}_1 < \mathtt{t}_2 \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{X}\Phi \mid \Phi\mathsf{U}\Phi,$$

where $\mathtt{t}, \mathtt{t}_1, \mathtt{t}_2 \in \mathrm{T}_{\mathrm{VAR}}$. The size of a formula is understood as its number of symbols with integers encoded with a binary representation. We use also the universal path quantifier A and the standard temporal connectives R and G ($\mathsf{A}\Phi \overset{\text{def}}{=} \neg\mathsf{E}\neg\Phi$, $\Phi_1\mathsf{R}\Phi_2 \overset{\text{def}}{=} \neg(\neg\Phi_1\mathsf{U}\,\neg\Phi_2)$, and $\mathsf{G}\Phi \overset{\text{def}}{=} \perp \mathsf{R}\,\Phi$ with $\perp$ equal to $\mathsf{E}(\mathtt{x} < \mathtt{x})$). No propositional variables occur in CTL*$(\mathbb{Z})$ formulae, but it is easy to simulate them with atomic formulae of the form $\mathsf{E}(\mathtt{x} = 0)$. We say that a formula in CTL*$(\mathbb{Z})$ is in *simple form* if it is in negation normal form (using A, R and $\vee$ as primitive) and all terms occurring in the formula are from $\mathrm{T}_{\mathrm{VAR}}^{\leqslant 1}$. State formulae are interpreted on worlds from a Kripke structure, whereas path formulae are interpreted on infinite paths. The two satisfaction relations are defined as follows (we omit the clauses for Boolean connectives), where $\mathcal{K} = (\mathcal{W}, \mathcal{R}, \mathfrak{v})$ is a total Kripke structure, and $w \in \mathcal{W}$.

- $\mathcal{K}, w \models \mathsf{E}\Phi \overset{\text{def}}{\Leftrightarrow}$ there is an infinite path π from w such that $\mathcal{K}, \pi \models \Phi$.

Let $\pi = w_0, w_1, \ldots$ be an infinite path of $\mathcal{K}$. Let us define $\mathfrak{v}(\pi, \mathsf{X}^j\mathtt{x}) \overset{\text{def}}{=} \mathfrak{v}(w_j, \mathtt{x})$, for all terms of the form $\mathsf{X}^j\mathtt{x}$. For every n, $\pi[n, +\infty)$ is the suffix of π truncated by the n first worlds.

- $\mathcal{K}, \pi \models \mathtt{t} = \mathfrak{d} \overset{\text{def}}{\Leftrightarrow} \mathfrak{v}(\pi, \mathtt{t}) = \mathfrak{d}$; $\mathcal{K}, \pi \models \mathtt{t}_1 \sim \mathtt{t}_2 \overset{\text{def}}{\Leftrightarrow} \mathfrak{v}(\pi, \mathtt{t}_1) \sim \mathfrak{v}(\pi, \mathtt{t}_2)$ for all $\sim \in \{<, =\}$,
- $\mathcal{K}, \pi \models \Phi\mathsf{U}\Psi \overset{\text{def}}{\Leftrightarrow}$ there is $j \geqslant 0$ such that $\mathcal{K}, \pi[j, +\infty) \models \Psi$ and for all $j' \in [0, j-1]$, we have $\mathcal{K}, \pi[j', +\infty) \models \Phi$;
- $\mathcal{K}, \pi \models \mathsf{X}\Phi \overset{\text{def}}{\Leftrightarrow} \mathcal{K}, \pi[1, +\infty) \models \Phi$.

Let us define two fragments of CTL*$(\mathbb{Z})$. Formulae in the logic CTL$(\mathbb{Z})$ are of the form

$$\phi := \mathsf{E}\ \Theta \mid \mathsf{A}\ \Theta \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{EX}\phi \mid \mathsf{E}\phi\mathsf{U}\phi \mid \mathsf{E}\phi\mathsf{R}\phi \mid \mathsf{AX}\phi \mid \mathsf{A}\phi\mathsf{U}\phi \mid \mathsf{A}\phi\mathsf{R}\phi,$$

where Θ is a constraint. LTL$(\mathbb{Z})$ formulae are defined from path formulae for CTL*$(\mathbb{Z})$ according to $\Phi := \Theta \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathsf{X}\Phi \mid \Phi\mathsf{U}\Phi \mid \Phi\mathsf{R}\Phi$, where Θ is a constraint. Negation occurs only in constraints since the LTL logical connectives have their dual in LTL$(\mathbb{Z})$. In contrast to CTL*$(\mathbb{Z})$ and CTL$(\mathbb{Z})$, LTL$(\mathbb{Z})$ formulae are evaluated over infinite paths of valuations $\mathfrak{v} : \mathrm{VAR} \to \mathbb{Z}$ (no branching involved).

The *satisfiability problem for* CTL*$(\mathbb{Z})$, written SAT(CTL*$(\mathbb{Z})$), is defined as follows.
Input: A CTL*$(\mathbb{Z})$ state formula ϕ.
Question: Is there a total Kripke structure $\mathcal{K}$ and a world w such that $\mathcal{K}, w \models \phi$?
The satisfiability problem SAT(CTL$(\mathbb{Z})$) for CTL$(\mathbb{Z})$ is defined analogously; for LTL$(\mathbb{Z})$, SAT(LTL$(\mathbb{Z})$) is the problem to decide whether there exists an infinite sequence of valuations $\mathfrak{v} : \mathrm{VAR} \to \mathbb{Z}$ for a given LTL$(\mathbb{Z})$ formula Φ.

Decidability, and, more precisely, PSPACE-completeness of SAT(LTL$(\mathbb{Z})$) is shown in [24]. For some strict fragments of CTL*$(\mathbb{Z})$, decidability is shown in [11, 12]. It is only recently in [14, 13, 15], that decidability is established for the full logic using a translation into a decidable second-order logic:

▶ **Proposition 1** ([14, 15]). SAT(CTL*$(\mathbb{Z})$) *is decidable.*

The proof in [14, 15] does not provide a complexity upper bound as the target decidable 2nd-order logic admits an automata-based decision procedure with open complexity [10, 8, 9].

Let us shortly explain why the satisfiability problem is challenging. First of all, observe that CTL*$(\mathbb{Z})$ has atomic formulae in which integer values at the current and successor states are compared. This prevents us from using a simple translation from CTL*$(\mathbb{Z})$ to CTL* with new propositions. Models of CTL*($\mathbb{Z}$) formulae can be viewed as an infinite network of constraints on $\mathbb{Z}$; even if a formula contains only a finite set of constants, a model may contain an infinite set of values, as it is the case for, e.g., the formula $\mathsf{EG}(\mathtt{x} < \mathsf{X}\mathtt{x})$. Hence a direct Boolean abstraction does not work. On the other hand, CTL*($\mathbb{Z}$) has no freeze quantifier and no data variable quantification, and hence no way to directly compare values at unbounded distance (but this can only be done by propagating local constraints), unlike e.g. the formalisms in [21, 63, 5, 1]. Hence, the lower bounds from [42] cannot apply either.

A problem related to satisfiability is the *model-checking problem*. Fragments of the model-checking problem involving a temporal logic similar to CTL*($\mathbb{Z}$) are investigated in [17, 11, 12, 34] (see also [39, 20, 70, 2]). However, model-checking problems with CTL*($\mathbb{Z}$)-like languages are easily undecidable, see e.g. [17, Theorem 1] and [54, Theorem 4.1] (more general constraints are used in [54] but undecidability proof uses only the constraints involved herein). The difference between model-checking and satisfiability is subtle and underlines that decidability/complexity of CTL($\mathbb{Z}$)/CTL*($\mathbb{Z}$) satisfiability is not immediate.

In this paper, we prove the precise computational complexity of SAT(CTL*($\mathbb{Z}$)) and SAT(CTL($\mathbb{Z}$)). We follow the automata-based approach, that is, we translate formulae in our logics into equivalent automata – tree constraint automata for CTL($\mathbb{Z}$), and Rabin tree constraint automata for CTL*($\mathbb{Z}$) – so that we can reduce the satisfiability problem for the logics to the nonemptiness problem for the corresponding automata.

3 Tree Constraint Automata

In this section, we introduce the class of tree constraint automata that accept sets of infinite trees of the form $\mathbb{t} : [0, D-1]^* \to (\Sigma \times \mathbb{Z}^{\beta})$ for some finite alphabet Σ and some $\beta \geqslant 1$. The transition relation of such automata states constraints between the β integer values at a node and the integer values at its children nodes. The acceptance condition is a Büchi condition (applied to the infinite branches of the input tree), but this can be easily extended to more general conditions (which we already consider by the end of this section). Moreover, our definition is specific to the concrete domain $\mathbb{Z}$ but it can be easily adapted to other concrete domains. Formally, a *tree constraint automaton* (TCA, for short) is a tuple $\mathbb{A} = (Q, \Sigma, D, \beta, Q_{\rm in}, \delta, F)$, where

- Q is a finite set of locations; Σ is a finite alphabet,
- $D \geqslant 1$ is the (branching) degree of (the trees accepted by) $\mathbb{A}$,
- $\beta \geqslant 1$ is the number of variables (a.k.a. registers),
- $Q_{\rm in} \subseteq Q$ is the set of initial locations; $F \subseteq Q$ encodes the Büchi acceptance condition,
- δ is a *finite* subset of $Q \times \Sigma \times (\mathrm{TreeCons}(\beta) \times Q)^D$, the transition relation. Here, TreeCons(β) denotes the constraints (Boolean combinations of atomic constraints) built over the terms $\mathtt{x}_1, \ldots, \mathtt{x}_{\beta}, \mathtt{x}'_1, \ldots, \mathtt{x}'_{\beta}$, where $\mathtt{x}'_i$ denotes the term $\mathsf{X}\mathtt{x}_i$. δ consists of tuples $(q, \mathtt{a}, (\Theta_0, q_0), \ldots, (\Theta_{D-1}, q_{D-1}))$, where $q \in Q$ is called the source location, $q_0, \ldots,$ $q_{D-1} \in Q$, $\mathtt{a} \in \Sigma$, and $\Theta_0, \ldots, \Theta_{D-1}$ are constraints.

Runs. Let $\mathbb{t} : [0, D-1]^* \to (\Sigma \times \mathbb{Z}^{\beta})$ be an infinite full D-ary tree over $\Sigma \times \mathbb{Z}^{\beta}$. A *run* of $\mathbb{A}$ on $\mathbb{t}$ is a mapping $\rho : [0, D-1]^* \to \delta$ satisfying the following conditions:

- $\rho(\varepsilon) = (q_{\rm in}, \ldots)$ such that $q_{\rm in} \in Q_{\rm in}$;
- for every $\mathbf{n} \in [0, D-1]^*$ with $\rho(\mathbf{n}) = (q, \mathtt{a}, (\Theta_0, q_0), \ldots, (\Theta_{D-1}, q_{D-1}))$, $\mathbb{t}(\mathbf{n} \cdot i) = (\mathtt{a}_i, \boldsymbol{z}_i)$, and $\rho(\mathbf{n} \cdot i)$ starts by the location q_i for all $0 \leqslant i < D$, we have $\mathbb{t}(\mathbf{n})$ of the form $(\mathtt{a}, \boldsymbol{z})$ and $\mathbb{Z} \models \Theta_i(\boldsymbol{z}, \boldsymbol{z_i})$ for all $0 \leqslant i < D$. Here, $\mathbb{Z} \models \Theta_i(\boldsymbol{z}, \boldsymbol{z_i})$ is a shortcut for $[\vec{\mathtt{x}} \leftarrow \boldsymbol{z}, \vec{\mathtt{x}'} \leftarrow \boldsymbol{z_i}] \models \Theta_i$ where $[\vec{\mathtt{x}} \leftarrow \boldsymbol{z}, \vec{\mathtt{x}'} \leftarrow \boldsymbol{z_i}]$ is a valuation $\mathfrak{v}$ on the variables $\{\mathtt{x}_j, \mathtt{x}'_j \mid j \in [1, \beta]\}$ with $\mathfrak{v}(\mathtt{x}_j) = \boldsymbol{z}(j)$ and $\mathfrak{v}(\mathtt{x}'_j) = \boldsymbol{z_i}(j)$ for all $j \in [1, \beta]$.

We show an example of a run ρ on $\mathbb{t}$ in Figure 1. Suppose ρ is a run of $\mathbb{A}$. Given a path $\pi = j_1 \cdot j_2 \cdot j_3 \ldots$ in ρ starting from ε, we define $\inf(\rho, \pi)$ to be the set of locations that appear infinitely often as the source locations of the transitions in $\rho(\varepsilon)\rho(j_1)\rho(j_1 \cdot j_2)\rho(j_1 \cdot j_2 \cdot j_3) \ldots$. A run ρ is *accepting* if for all paths π in ρ starting from ε, we have $\inf(\rho, \pi) \cap F \neq \emptyset$. We write $\mathrm{L}(\mathbb{A})$ to denote the set of trees $\mathbb{t}$ that admit an accepting run.

Nonemptiness problem. As usual, the *nonemptiness problem for TCA* asks whether a TCA $\mathbb{A}$ satisfies $\mathrm{L}(\mathbb{A}) \neq \emptyset$. To define the size of $\mathbb{A}$ in a reasonably succinct encoding, we need to consider the size of constraints from $\mathrm{TreeCons}(\beta)$. Indeed, unlike (plain) Büchi tree automata [68], the number of transitions in a tree constraint automaton is *a priori* unbounded ($\mathrm{TreeCons}(\beta)$ is infinite) and the maximal size of a constraint occurring in transitions is unbounded too. In particular, this means that $\mathrm{card}(\delta)$ is a priori unbounded, even if Q and Σ are fixed. We write $\mathtt{MCS}(\mathbb{A})$ to denote the maximal size of a constraint occurring in $\mathbb{A}$ (with binary encoding of the integers). The complexity of the nonemptiness problem should take into account these parameters. Note also that our automaton model differs from the Presburger Büchi tree automata from [62, 6] for which, in the runs, arithmetical expressions are related to constraints between numbers of children labelled by different locations. Herein, the arithmetical expressions state constraints between integer values.

Next, we introduce a variant of TCA by considering the Rabin acceptance condition (as opposed to the Büchi acceptance condition). A *Rabin tree constraint automaton* (Rabin TCA, for short) is a tuple $\mathbb{A} = (Q, \Sigma, D, \beta, Q_{\mathrm{in}}, \delta, \mathcal{F})$ defined as for TCA except that $\mathcal{F}$ is a set of pairs of the form (L, U), where $L, U \subseteq Q$. All the definitions about TCA apply except that a run $\rho : [0, D-1]^* \to \delta$ is *accepting* iff for all paths π in ρ starting from ε, there is some $(L, U) \in \mathcal{F}$ such that $\inf(\rho, \pi) \cap L \neq \emptyset$ and $\inf(\rho, \pi) \cap U = \emptyset$.

Finite alphabet. The set Σ in data trees $\mathbb{t} : [0, D-1]^* \to (\Sigma \times \mathbb{Z}^{\beta})$ plays no specific role herein, especially that it could be encoded with simple constraints of the form $\mathtt{x}^{\star} = \mathfrak{d}$, where $\mathtt{x}^{\star}$ is a distinguished variables. Its inclusion is more handy when the logical atomic formulae include constraints on variables *and* propositional variables, as done in [27, Section 5.2] dedicated to description logics (developments on description logics are very little in this paper, due to lack of space).

4 Complexity of the Nonemptiness Problem for TCA

This section is dedicated to prove the ExpTime-completeness of the nonemptiness problem for TCA (Theorem 11) and Rabin TCA (Theorem 13) (we make a distinction between TCA and Rabin TCA because the complexity bounds differ slightly, see Lemma 10 and Lemma 12). Before we prove the ExpTime upper bound, let us drop a few words on the lower bound. We show ExpTime-hardness of the nonemptiness problem for TCA by reduction from the acceptance problem for alternating Turing machines running in polynomial space, see e.g. [18, Corollary 3.6]. Indeed, the polynomial-space tape using a finite alphabet Σ can be encoded by a polynomial amount of variables taking values in $[1, \mathrm{card}(\Sigma)]$, details can be found in [27, Section 4.1]. ExpTime-hardness for Rabin TCA follows, as every TCA with set F of accepting locations can be encoded as a Rabin TCA with a single Rabin pair $(F, \emptyset)$.

The proof of the ExpTime upper bound is divided into two parts. In order to determine whether $\mathrm{L}(\mathbb{A})$ is nonempty for a given TCA $\mathbb{A}$, we first reduce the existence of some tree $\mathbb{t} \in \mathrm{L}(\mathbb{A})$ to the existence of some *regular symbolic tree* that is *satisfiable*, that is, it admits a concrete model (Sections 4.1 and 4.2). Second, we characterise the complexity of determining the existence of such satisfiable regular symbolic trees (Section 4.3). The result for Rabin TCA is presented in Section 4.4.

From now on, we assume a fixed TCA $\mathbb{A} = (Q, \Sigma, D, \beta, Q_{\rm in}, \delta, F)$ with the constants $\mathfrak{d}_1, \ldots, \mathfrak{d}_\alpha$ occurring in $\mathbb{A}$ such that $\mathfrak{d}_1 < \cdots < \mathfrak{d}_\alpha$ (we assume there is at least one constant).

4.1 Symbolic Trees

A *type* over the variables $\mathtt{z}_1, \ldots, \mathtt{z}_n$ is an expression of the form

$$(\bigwedge_i \Theta_i^{\rm CST}) \wedge (\bigwedge_{i<j} \mathtt{z}_i \sim_{i,j} \mathtt{z}_j), \text{ where}$$

- for all $i \in [1, n]$, $\Theta_i^{\rm CST}$ is equal to either $\mathtt{z}_i < \mathfrak{d}_1$, or $\mathtt{z}_i > \mathfrak{d}_\alpha$ or $\mathtt{z}_i = \mathfrak{d}$ for some $\mathfrak{d} \in [\mathfrak{d}_1, \mathfrak{d}_\alpha]$. This definition goes a bit beyond the constraint language in $\mathbb{Z}$ (because of expressions of the form $\mathtt{z}_i < \mathfrak{d}_1$ and $\mathtt{z}_i > \mathfrak{d}_\alpha$), but this is harmless in the sequel. What really matters in a type is the way the variables are compared to each other and to the constants.
- $\sim_{i,j} \in \{>, =, <\}$ for all $i < j$.

Checking the satisfiability of a type can be done in polynomial-time, based on a standard cycle detection, see e.g. [17, Lemma 5.5]. The set of *satisfiable types* built over the variables $\mathtt{x}_1, \ldots, \mathtt{x}_\beta, \mathtt{x}'_1, \ldots, \mathtt{x}'_\beta$ is written $\mathrm{SatTypes}(\beta)$ (n above is equal here to 2β). Observe that $\mathrm{card}(\mathrm{SatTypes}(\beta)) \leqslant ((\mathfrak{d}_\alpha - \mathfrak{d}_1) + 3)^{2\beta} \times 3^{\beta^2}$. The *restriction* of the type Θ to some set of variables $X \subseteq \{\mathtt{x}_i, \mathtt{x}'_i \mid i \in [1, \beta]\}$ is made of all the conjuncts in which only variables in X occur. The type Θ restricted to $\{\mathtt{x}'_i \mid i \in [1, \beta]\}$ *agrees* with the type Θ' restricted to $\{\mathtt{x}_i \mid i \in [1, \beta]\}$ iff Θ and Θ' are logically equivalent modulo the renaming for which $\mathtt{x}_i$ and $\mathtt{x}'_i$ are substituted, for all $i \in [1, \beta]$. For instance, in Figure 1, Θ restricted to $\{\mathtt{x}'_1, \mathtt{x}'_2\}$ agrees with Θ_0 restricted to $\{\mathtt{x}_1, \mathtt{x}_2\}$. The main properties we use about satisfiable types are stated below.

(I) Let $\boldsymbol{z}, \boldsymbol{z}' \in \mathbb{Z}^\beta$. There is a unique $\Theta \in \mathrm{SatTypes}(\beta)$ such that $\mathbb{Z} \models \Theta(\boldsymbol{z}, \boldsymbol{z}')$.

(II) For every constraint Θ built over the variables $\mathtt{x}_1, \ldots, \mathtt{x}_\beta, \mathtt{x}'_1, \ldots, \mathtt{x}'_\beta$ and the constants $\mathfrak{d}_1, \ldots, \mathfrak{d}_\alpha$ there is a disjunction $\Theta_1 \vee \cdots \vee \Theta_\gamma$ logically equivalent to Θ and each Θ_i belongs to $\mathrm{SatTypes}(\beta)$ (empty disjunction stands for $\perp$).

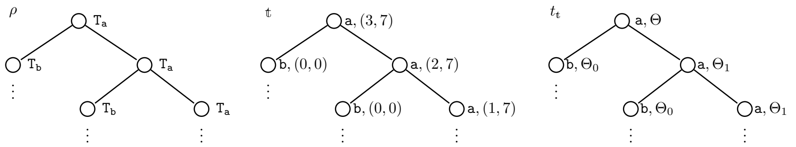
(III) For all $\Theta \neq \Theta' \in \mathrm{SatTypes}(\beta)$, the constraint $\Theta \wedge \Theta'$ is not satisfiable.

The proof is by an easy verification and this justifies the term “type” used in this context.

Abstraction with types. A *symbolic tree* t is a map $t : [0, D-1]^* \rightarrow \Sigma \times \mathrm{SatTypes}(\beta)$. Symbolic trees are intended to be abstractions of trees labelled with concrete values in $\mathbb{Z}$. Given a tree $\mathfrak{t} : [0, D-1]^* \rightarrow \Sigma \times \mathbb{Z}^\beta$, its *abstraction* is the symbolic tree $t_{\mathfrak{t}} : [0, D-1]^* \rightarrow \Sigma \times \mathrm{SatTypes}(\beta)$ such that for all $\mathbf{n} \cdot i \in [0, D-1]^*$ with $\mathfrak{t}(\mathbf{n}) = (\mathtt{a}, \boldsymbol{z})$ and $\mathfrak{t}(\mathbf{n} \cdot i) = (\mathtt{a}_i, \boldsymbol{z}_i)$, $t_{\mathfrak{t}}(\mathbf{n} \cdot i) \overset{\rm def}{=} (\mathtt{a}_i, \Theta_i)$ for the unique $\Theta_i \in \mathrm{SatTypes}(\beta)$ such that $\mathbb{Z} \models \Theta_i(\boldsymbol{z}, \boldsymbol{z}_i)$. Note that the primed variables in Θ_i refer to the β values at the node $\mathbf{n} \cdot i$, whereas the unprimed ones refer to the β values at the parent node $\mathbf{n}$. At the root ε with $\mathfrak{t}(\varepsilon) = (\mathtt{a}, \boldsymbol{z})$, we have $t_{\mathfrak{t}}(\varepsilon) \overset{\rm def}{=} (\mathtt{a}, \Theta)$ for the unique $\Theta \in \mathrm{SatTypes}(\beta)$ such that $\mathbb{Z} \models \Theta(\mathbf{0}, \boldsymbol{z})$, where $\mathbf{0} \in \mathbb{Z}^\beta$ is arbitrary as there are actually no parent values at the root. A symbolic tree t is *satisfiable* $\overset{\rm def}{\Leftrightarrow}$ there is $\mathfrak{t} : [0, D-1]^* \rightarrow \Sigma \times \mathbb{Z}^\beta$ such that $t_{\mathfrak{t}} = t$. We say that $\mathfrak{t}$ *witnesses the satisfaction of* t, also written $\mathfrak{t} \models t$. A symbolic tree t is *regular* if its set of subtrees is finite.

$\mathbb{A}$-consistency. In our quest to decide whether $\mathrm{L}(\mathbb{A}) \neq \emptyset$, we are interested in symbolic trees that satisfy certain properties that we subsume under the name $\mathbb{A}$-*consistent*. A symbolic tree $t : [0, D-1]^* \rightarrow \Sigma \times \mathrm{SatTypes}(\beta)$ is $\mathbb{A}$-*consistent* if the following conditions are satisfied:

- t is *locally consistent*: for every node $\mathbf{n}$, the type Θ labelling $\mathbf{n}$ restricted to $\mathtt{x}'_1, \ldots, \mathtt{x}'_\beta$ agrees with all types Θ_i labelling its children nodes $\mathbf{n} \cdot i$ restricted to $\mathtt{x}_1, \ldots, \mathtt{x}_\beta$, and



$$\begin{aligned}
\Theta &\stackrel{\text{def}}{=} \mathfrak{d}_1 = \mathtt{x}_1 = \mathtt{x}_2 < \underline{\mathtt{x}'_1 < \mathtt{x}'_2}\\
\Theta_0 &\stackrel{\text{def}}{=} \mathfrak{d}_1 = \mathtt{x}'_1 = \mathtt{x}'_2 < \underline{\mathtt{x}_1 < \mathtt{x}_2}\\
\Theta_1 &\stackrel{\text{def}}{=} \mathfrak{d}_1 < \mathtt{x}'_1 < \underline{\mathtt{x}_1 < \mathtt{x}_2} = \mathtt{x}'_2
\end{aligned}$$

Figure 1 A tree $\mathbb{t}$ (middle), a run ρ of some TCA on $\mathbb{t}$ (left), where, $\mathtt{T}_\mathtt{a} = (q, \mathtt{a}, (\Theta_0, q), (\Theta_1, q))$ and $\mathtt{T}_\mathtt{b} = (q, \mathtt{b}, (\Theta_0, q), (\Theta_1, q))$, and the symbolic tree $t_\mathbb{t}$ (abstraction of $\mathbb{t}$) (right).

- there is an accepting run ρ of $\mathbb{A}$ (but ignoring the conditions on data values) such that for all $\mathbf{n} \in [0, D-1]^*$ with $t(\mathbf{n}) = (\mathtt{a}, \Theta)$, $t(\mathbf{n} \cdot i) = (\mathtt{a}_i, \Theta_i)$ for all $i \in [0, D-1]$, and $\rho(\mathbf{n}) = (q, \mathtt{a}, (\Theta'_0, q_0) \ldots (\Theta'_{D-1}, q_{D-1}))$, we have $\Theta_i \models \Theta'_i$ for all $i \in [0, D-1]$.

▶ **Example 2.** In Figure 1, we show a tree $\mathbb{t}$ with concrete values in $\mathbb{Z}^\beta$ for $\beta = 2$ (middle) and its abstraction $t_\mathbb{t}$ (right). We assume that $\mathfrak{d}_1 = 0$ is the only constant; consequently, $t_\mathbb{t}$ uses constraints in SatTypes(β) that are built with variables $\mathtt{x}_1, \mathtt{x}_2$, their primed variants $\mathtt{x}'_1, \mathtt{x}'_2$, and the constant $\mathfrak{d}_1$. We underline constraints to illustrate the property of local consistency.

It is not hard to prove that the set of all $\mathbb{A}$-consistent symbolic trees is ω-regular, that is, it can be accepted by a classical tree automaton without constraints. In the following, we use the standard letter A to distinguish automata *without constraints* from TCA.

▶ **Lemma 3.** *There exists a Büchi tree automaton (without constraints) $A_{cons(\mathbb{A})}$ such that $\mathrm{L}(A_{cons(\mathbb{A})})$ is equal to the set of $\mathbb{A}$-consistent symbolic trees.*

The locations in $A_{\mathrm{cons}(\mathbb{A})}$ are from SatTypes(β) × Q and the transition relation for $A_{\mathrm{cons}(\mathbb{A})}$ can be decided in polynomial-time in $\mathrm{card}(\delta) + \beta + \mathrm{card}(\Sigma) + D + \mathtt{MCS}(\mathbb{A})$.

However, not every $\mathbb{A}$-consistent symbolic tree admits a concrete model. Thus the more important property is to check whether $\mathrm{L}(A_{\mathrm{cons}(\mathbb{A})})$ contains some *satisfiable* symbolic tree (and we explain how to do this in the next two subsections). The result below is a variant of many similar results relating symbolic models and concrete models in logics for concrete domains, see e.g. [23, Corollary 4.1], [38, Lemma 3.4], [16, Theorem 25], [46, Theorem 11].

▶ **Lemma 4.** $\mathrm{L}(\mathbb{A}) \neq \varnothing$ *iff there is a* satisfiable *symbolic tree in* $\mathrm{L}(A_{cons(\mathbb{A})})$.

4.2 Satisfiability for Regular Locally Consistent Symbolic Trees

Below, we focus on deciding when $\mathrm{L}(A_{\mathrm{cons}(\mathbb{A})})$ contains a *satisfiable* symbolic tree, while evaluating the complexity to check its existence. Given a locally consistent symbolic tree $t : [0, D-1]^* \to \Sigma \times \mathrm{SatTypes}(\beta)$, we introduce an infinite labelled graph that contains exactly the same types as t but expressed in a tree-like graph from which it is convenient to characterize satisfiability in terms of paths, under the premise that t is regular. Similar symbolic structures are introduced in [49, 23, 14, 46]. The graph is equal to the structure

$$G_t^{\mathrm{C}} = (V_t, \xrightarrow{=}, \xrightarrow{<}, U_{<\mathfrak{d}_1}, (U_i)_{i \in [\mathfrak{d}_1, \mathfrak{d}_\alpha]}, U_{>\mathfrak{d}_\alpha}),$$

where $V_t = [0, D-1]^* \times (\{\mathtt{x}_1, \ldots, \mathtt{x}_\beta\} \cup \{\mathfrak{d}_1, \mathfrak{d}_\alpha\})$, $\xrightarrow{=}$ and $\xrightarrow{<}$ are two binary relations over V_t, and $\{U_{<\mathfrak{d}_1}, U_{\mathfrak{d}_1}, U_{\mathfrak{d}_1+1}, \ldots, U_{\mathfrak{d}_\alpha}, U_{>\mathfrak{d}_\alpha}\}$ is a partition of V_t. Elements in $\{\mathtt{x}_1, \ldots, \mathtt{x}_\beta\} \cup \{\mathfrak{d}_1, \mathfrak{d}_\alpha\}$ are denoted by $\mathtt{xd}, \mathtt{xd}_1, \mathtt{xd}_2, \ldots$ (variables or constants). Moreover, $V_t^\beta \stackrel{\text{def}}{=} [0, D-1]^* \times \{\mathtt{x}_1, \ldots, \mathtt{x}_\beta\}$. The rationale behind the construction of G_t^{C} is to reflect the constraints between parent and children nodes as well as the constraints regarding constants, in such a way that, if $\mathbb{t}$ witnesses the satisfaction of t, then, e.g., $\mathbb{t}(\mathbf{n})(\mathtt{xd}) < \mathbb{t}(\mathbf{n}')(\mathtt{xd}')$ if $(\mathbf{n}, \mathtt{xd}) \xrightarrow{<} (\mathbf{n}', \mathtt{xd}')$, and $\mathbb{t}(\mathbf{n})(\mathtt{xd}) = \mathfrak{d}_1$ if $(\mathbf{n}, \mathtt{xd}) \in U_{\mathfrak{d}_1}$. Here are all conditions for building G_t^{C}.

(VAR) For all $(\mathbf{n}, \mathtt{x}_i), (\mathbf{n}', \mathtt{x}_{i'}) \in V_t^{\beta}$, for all $\sim \in \{<, =\}$, $(\mathbf{n}, \mathtt{x}_i) \xrightarrow{\sim} (\mathbf{n}', \mathtt{x}_{i'})$ iff either $\mathbf{n}' = \mathbf{n} \cdot j$ and $\mathtt{x}_i \sim \mathtt{x}'_{i'}$ in Θ with $t(\mathbf{n}') = (\mathtt{a}, \Theta)$, or $\mathbf{n} = \mathbf{n}'$ and $\mathtt{x}'_i \sim \mathtt{x}'_{i'}$ in Θ with $t(\mathbf{n}') = (\mathtt{a}, \Theta)$, or $\mathbf{n} = \mathbf{n}' \cdot j$ and $\mathtt{x}'_i \sim \mathtt{x}_{i'}$ in Θ with $t(\mathbf{n}) = (\mathtt{a}, \Theta)$.

(P1) For all $\mathfrak{d} \in [\mathfrak{d}_1, \mathfrak{d}_\alpha]$ and $(\mathbf{n}, \mathtt{x}_j) \in V_t^{\beta}$, $(\mathbf{n}, \mathtt{x}_j) \in U_{\mathfrak{d}}$ iff $\mathtt{x}'_j = \mathfrak{d}$ in Θ with $t(\mathbf{n}) = (\mathtt{a}, \Theta)$.

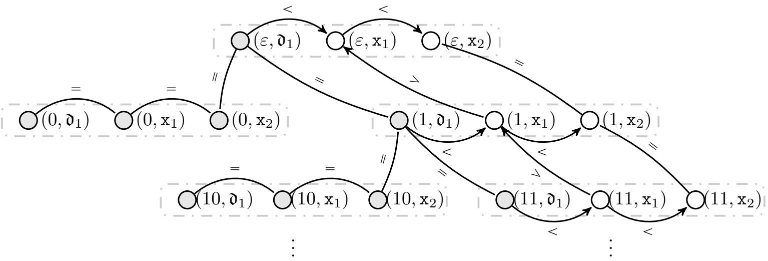
(P2) For all $(\mathbf{n}, \mathtt{x}_j) \in V_t^{\beta}$, $(\mathbf{n}, \mathtt{x}_j) \in U_{<\mathfrak{d}_1}$ iff $\mathtt{x}'_j < \mathfrak{d}_1$ in Θ with $t(\mathbf{n}) = (\mathtt{a}, \Theta)$.

(P3) For all $(\mathbf{n}, \mathtt{x}_j) \in V_t^{\beta}$, $(\mathbf{n}, \mathtt{x}_j) \in U_{>\mathfrak{d}_\alpha}$ iff $\mathtt{x}'_j > \mathfrak{d}_\alpha$ in Θ with $t(\mathbf{n}) = (\mathtt{a}, \Theta)$.

(P4) For all $\mathbf{n} \in [0, D-1]^*$, $(\mathbf{n}, \mathfrak{d}_1) \in U_{\mathfrak{d}_1}$ and $(\mathbf{n}, \mathfrak{d}_\alpha) \in U_{\mathfrak{d}_\alpha}$.

(CONS) This is about elements of V_t labelled by constants and how the edge labels reflect the relationships between the constants. Formally, for all $((\mathbf{n}, \mathtt{xd}), (\mathbf{n}', \mathtt{xd}')) \in (V_t \times V_t) \backslash (V_t^{\beta} \times V_t^{\beta})$, for all $\mathfrak{d}^{\dagger}, \mathfrak{d}^{\dagger\dagger}$ in "$< \mathfrak{d}_1$", $\mathfrak{d}_1, \ldots, \mathfrak{d}_\alpha$, "$> \mathfrak{d}_\alpha$" s.t. $(\mathbf{n}, \mathtt{xd}) \in U_{\mathfrak{d}^{\dagger}}$ and $(\mathbf{n}', \mathtt{xd}') \in U_{\mathfrak{d}^{\dagger\dagger}}$, for all $\sim \in \{<, =\}$, $(\mathbf{n}, \mathtt{xd}) \xrightarrow{\sim} (\mathbf{n}', \mathtt{xd}')$ iff either $\mathfrak{d}^{\dagger}, \mathfrak{d}^{\dagger\dagger} \in [\mathfrak{d}_1, \mathfrak{d}_\alpha]$ and $\mathfrak{d}^{\dagger} \sim \mathfrak{d}^{\dagger\dagger}$, or $\mathfrak{d}^{\dagger} =$ "$< \mathfrak{d}_1$", $\mathfrak{d}^{\dagger\dagger} \neq$ "$< \mathfrak{d}_1$" and $\sim$ is equal to $<$ or $\mathfrak{d}^{\dagger} \neq$ "$> \mathfrak{d}_\alpha$", $\mathfrak{d}^{\dagger\dagger} =$ "$> \mathfrak{d}_\alpha$" and $\sim$ is equal to $<$.

Below, we illustrate the definition of the graph $G^{\mathrm{C}}_{t_{\mathfrak{t}}}$ for the symbolic tree $t_{\mathfrak{t}}$ in Figure 1. The edges labelled with $=$ or $<$ reflect the constraints (we omit edges if they can be inferred from the other edges). For instance, $(1, \mathtt{x}_1) \xrightarrow{<} (\varepsilon, \mathtt{x}_1)$ corresponds to the constraint $\mathtt{x}'_1 < \mathtt{x}_1$. Grey nodes are in $U_{\mathfrak{d}_1}$, all other nodes are in $U_{>\mathfrak{d}_1}$ (no nodes in $U_{<\mathfrak{d}_1}$).



A map $p : \mathbb{N} \to V_t$ is a *path map* in $G^{\mathrm{C}}_t \overset{\text{def}}{\Leftrightarrow}$ for all $i \in \mathbb{N}$, either $p(i) \xrightarrow{=} p(i+1)$ or $p(i) \xrightarrow{<} p(i+1)$ in G^{C}_t. Similarly, $r : \mathbb{N} \to V_t$ is a *reverse path map* in $G^{\mathrm{C}}_t \overset{\text{def}}{\Leftrightarrow}$ for all $i \in \mathbb{N}$, either $r(i) \xrightarrow{=} r(i+1)$ or $r(i+1) \xrightarrow{<} r(i)$. A path map p (resp. reverse path map r) is *strict* $\overset{\text{def}}{\Leftrightarrow}$ $\{i \in \mathbb{N} \mid p(i) \xrightarrow{<} p(i+1)\}$ (resp. $\{i \in \mathbb{N} \mid r(i+1) \xrightarrow{<} r(i)\}$) is infinite. An *infinite branch* $\mathcal{B}$ is an element of $[0, D-1]^{\omega}$. We write $\mathcal{B}[i, j]$ with $i \leqslant j$ to denote the subsequence $\mathcal{B}(i) \cdot \mathcal{B}(i+1) \cdots \mathcal{B}(j)$. Given $(\mathbf{n}, \mathtt{xd}) \in V_t$, a path map p *from* $(\mathbf{n}, \mathtt{xd})$ *along* $\mathcal{B}$ is such that $p(0) = (\mathbf{n}, \mathtt{xd})$ and for all $i \geqslant 0$, $p(i)$ is of the form $(\mathbf{n} \cdot \mathcal{B}[0, i], \cdot)$. A reverse path map r from $(\mathbf{n}, \mathtt{xd})$ along $\mathcal{B}$ admits a similar definition. We present the condition $(\bigstar^{\mathrm{C}})$ that is the central property for characterising regular symbolic trees in $\mathrm{L}(\mathcal{A}_{\mathrm{cons}(\mathbb{A})})$ that are satisfiable, following the remarkable result established in [46, Lemma 22] that non-satisfiability of a symbolic tree can be witnessed along a *single branch*.

$(\bigstar^{\mathrm{C}})$ There are *no* elements $(\mathbf{n}, \mathtt{xd})$, $(\mathbf{n}, \mathtt{xd}')$ in G^{C}_t (same node $\mathbf{n}$ from $[0, D-1]^*$) and no infinite branch $\mathcal{B}$ such that

1. there exists a path map p from $(\mathbf{n}, \mathtt{xd})$ along $\mathcal{B}$,
2. there exists a reverse path map r from $(\mathbf{n}, \mathtt{xd}')$ along $\mathcal{B}$,
3. p or r is strict, and
4. for all $i \in \mathbb{N}$, $p(i) \xrightarrow{<} r(i)$.

The following proposition states a key property: non-satisfaction of a regular locally consistent symbolic tree can be witnessed along a *single* branch by violation of $(\bigstar^{\mathrm{C}})$.

▶ **Proposition 5.** *For every regular locally consistent symbolic tree t, G^{C}_t satisfies $(\bigstar^{C})$ iff t is satisfiable.*

A proof can be found in [27, Section 7].

▶ **Example 6.** Assume that every node along the rightmost branch in the symbolic tree $t_{\mathfrak{t}}$ in Figure 1 is labelled with (a, Θ_1). Then $t_{\mathfrak{t}}$ is not satisfiable: in order to satisfy Θ_1's conjunct $\mathtt{x}'_1 < \mathtt{x}_1$, the value of $\mathtt{x}_1$ must inevitably become finally smaller than $\mathfrak{d}_1$, violating the conjunct $\mathfrak{d}_1 < \mathtt{x}_1$. Consequently, the rightmost branch of $G^{\mathrm{C}}_{t_{\mathfrak{t}}}$ presented above does not satisfy $(\bigstar^{\mathrm{C}})$: there exists a path map p from $(\varepsilon, \mathfrak{d}_1)$ along 1^ω, there exists a strict reverse path map r from $(\varepsilon, \mathtt{x}_1)$ along 1^ω, and for all $i \in \mathbb{N}$ we have $p(i) \xrightarrow{\leq} r(i)$.

New constant nodes. Proposition 5 above is a variant of [46, Lemma 22]. Before going any further, let us in short explain the improvement of our developments compared to what is done in [46, 45]. The *framified constraint graphs* defined in [46, Definition 14] correspond to the above defined graph G^{C}_t without $[0, D-1]^* \times \{\mathfrak{d}_1, \mathfrak{d}_\alpha\}$ and corresponding edges. However, Example 6 illustrates the importance of taking into account these elements when deciding satisfiability (without $\mathfrak{d}_1$, the graph would satisfy $(\bigstar^{\mathrm{C}})$). Actually, Example 6 invalidates $(\bigstar)$ as used in [46, 45] because the constants are missing to apply properly [15]. The problematic part in [46, 45] is due to the proof of [45, Lemma 5.18] whose main argument takes advantage of [15] but without the elements related to constant values (see also [24, Lemma 8]). With Proposition 5, we also propose a proof to characterise satisfiability of symbolic trees that is independent of [15]. Note also that the condition $(\bigstar)$ in [46, Section 3.3] generalises the condition $C_{\mathbb{Z}}$ from [23, Section 6] (see also the condition $\mathcal{C}$ in [24, Definition 2] and a similar condition in [32, Section 2]). A condition similar to $(\bigstar)$ is also introduced recently in [7, Lemma 18] to decide a realizability problem based on $\mathrm{LTL}(\mathbb{Z}, <, =)$.

We recall that there are *nonregular* locally consistent symbolic trees t such that G^{C}_t satisfies $(\bigstar^{\mathrm{C}})$ (see e.g. [23, 46]) but t is not satisfiable; indeed, satisfiability of symbolic trees is not an ω-regular property. The next result states that $(\bigstar^{\mathrm{C}})$ is ω-regular; hence, satisfiability of symbolic trees can be overapproximated advantageously.

▶ **Lemma 7.** *There is a Rabin tree automaton $A_{\bigstar^C}$ such that $\mathrm{L}(A_{\bigstar^C}) = \{t \mid G^C_t \text{ satisfies } (\bigstar^C)\}$, the number of Rabin pairs is bounded above by $8(\beta + 2)^2 + 3$, the number of locations is exponential in β, the transition relation can be decided in polynomial-time in*

$$\max(\lceil log(|\mathfrak{d}_1|)\rceil, \lceil log(|\mathfrak{d}_\alpha|)\rceil) + \beta + card(\Sigma) + D.$$

Proof sketch. The proof of Lemma 7 is structured as follows (see [27, Section 4.3]). (1) We construct a Büchi word automaton A_B accepting the complement of $(\bigstar^{\mathrm{C}})$ for $D = 1$. (2) A_B is nondeterministic, but we can determinize it and get a deterministic Rabin word automaton $A_{B \to R}$ such that $\mathrm{L}(A_B) = \mathrm{L}(A_{B \to R})$ (using the determinisation construction from [60, Theorem 1.1]). (3) By an easy construction, we obtain a deterministic Street word automaton A_S accepting the complement of $\mathrm{L}(A_{B \to R})$; it accepts words that satisfy $(\bigstar^{\mathrm{C}})$ for $D = 1$. (4) By [60, Lemma 1.2], we construct a deterministic Rabin word automaton A_R s.t. $\mathrm{L}(A_S) = \mathrm{L}(A_R)$. (5) Finally, we construct a Rabin tree automaton $A_{\bigstar^{\mathrm{C}}}$, the intuitive idea is to "let run the automaton A_R" along every branch of a run of $A_{\bigstar^{\mathrm{C}}}$, doable thanks to the determinism of A_R. Since $(\bigstar^{\mathrm{C}})$ states a property on every branch, we are done. ◀

Differences with [46]. Lemma 7 is similar to [46, Proposition 26] but there is an essential difference: the number of Rabin pairs in Lemma 7 is not a constant but a value depending on β, an outcome of our investigations. It is important to know the number of Rabin pairs in $A_{\bigstar^{\mathrm{C}}}$ for our complexity analysis as checking nonemptiness of Rabin tree automata is *exponential* in the number of Rabin pairs [30, Theorem 4.1]. Our proof of Lemma 7 also proposes a slight novelty compared to the construction in [46]: we design $A_{\bigstar^{\mathrm{C}}}$ without firstly constructing *a tree automaton for the complement language* (as done in [46]) and then using results from [55] (elimination of alternation in tree automata). Our new approach shall be rewarding: not only we can better understand how to express the condition ($\bigstar^{\mathrm{C}}$), but also we control the size parameters of $A_{\bigstar^{\mathrm{C}}}$ involved in our forthcoming complexity analysis. Furthermore, it may be useful to implement the decision procedure for solving the satisfiability problem for CTL($\mathbb{Z}$) (resp. for CTL*($\mathbb{Z}$)). Note also that the above analysis about the number of Rabin pairs is independent from the question discussed above about having the elements in $[0, D-1]^* \times \{\mathfrak{d}_1, \mathfrak{d}_\alpha\}$ within G_t^{C}.

Summarizing the developments so far, we can conclude this subsection as follows:

▶ **Lemma 8.** $\mathrm{L}(\mathbb{A}) \neq \emptyset$ *iff* $\mathrm{L}(A_{cons(\mathbb{A})}) \cap \mathrm{L}(A_{\bigstar^{C}}) \neq \emptyset$.

For its proof, by way of example, if $\mathrm{L}(A_{\mathrm{cons}(\mathbb{A})}) \cap \mathrm{L}(A_{\bigstar^{\mathrm{C}}})$ is non-empty, then as $\mathrm{L}(A_{\mathrm{cons}(\mathbb{A})}) \cap \mathrm{L}(A_{\bigstar^{\mathrm{C}}})$ is regular, it contains a regular $\mathbb{A}$-consistent symbolic tree t (see e.g. [58] and [64, Section 6.3] for the existence of regular trees) and by Proposition 5, t is satisfiable. By Lemma 4, we get $\mathrm{L}(\mathbb{A}) \neq \emptyset$. For the other direction, we use Lemma 3 as well as the property that for every satisfiable symbolic tree t, G_t^{C} satisfies the condition ($\bigstar^{\mathrm{C}}$).

4.3 ExpTime Upper Bound for TCAs

Lemma 8 justifies why deciding the nonemptiness of $\mathrm{L}(A_{\mathrm{cons}(\mathbb{A})}) \cap \mathrm{L}(A_{\bigstar^{\mathrm{C}}})$ is crucial. In the proof of Lemma 9 below (see [27, Section 4.4]), we propose a construction for the intersection of Rabin tree automata that only performs an exponential blow-up for the number of locations, which is fine for our purposes.

▶ **Lemma 9.** *There is a Rabin tree automaton A such that $\mathrm{L}(A) = \mathrm{L}(A_{cons(\mathbb{A})}) \cap \mathrm{L}(A_{\bigstar^{C}})$ and the number of Rabin pairs is polynomial in β, the number of locations is in $\mathcal{O}(card(\mathrm{SatTypes}(\beta)) \times card(Q) \times 2^{P(\beta)})$ for some polynomial $P(\cdot)$ and the transition relation can be decided in polynomial-time in $card(\delta) + \beta + card(\Sigma) + D + \mathtt{MCS}(\mathbb{A})$.*

Nonemptiness of Rabin tree automata is polynomial in the cardinality of the transition relation and exponential in the number of Rabin pairs, see e.g. [30, Theorem 4.1]. More precisely, it is in time $(m \times n)^{\mathcal{O}(n)}$, where m is the number of locations and n is the number of Rabin pairs, see the statement [30, Theorem 4.1]. However, this is not exactly what we need herein, as the complexity expression above concerns *binary* trees, and it assumes that the transition relation δ can be decided in constant time. If, as in our case, $D \geqslant 1$ and deciding whether a tuple belongs to δ requires γ time units, checking nonemptiness is actually in time $(\mathrm{card}(\delta) \times \gamma \times n)^{\mathcal{O}(n)}$ (by scrutiny of the proof of [30, Theorem 4.1], page 144). Here, γ may depend on parameters related to $\mathbb{A}$ and in Lemma 10 below, γ takes the value $\mathrm{card}(\delta) + \beta + \mathrm{card}(\Sigma) + D + \mathtt{MCS}(\mathbb{A})$ (by Lemma 9). Hence the following result:

▶ **Lemma 10.** *The nonemptiness problem for TCA can be solved in time in $\mathcal{O}(R_1\big(card(Q) \times card(\delta) \times \mathtt{MCS}(\mathbb{A}) \times card(\Sigma) \times R_2(\beta)\big)^{R_2(\beta) \times R_3(D)})$ for some polynomials R_1, R_2 and R_3.*

Assuming that the size of the TCA $\mathbb{A} = (Q, \Sigma, D, \beta, Q_{\mathrm{in}}, \delta, F)$, written $\mathtt{size}(\mathbb{A})$, is polynomial in $\mathrm{card}(Q) + \mathrm{card}(\delta) + D + \beta + \mathtt{MCS}(\mathbb{A})$ (which makes sense for a reasonably succinct encoding), from the computation of the bound in Lemma 10, the nonemptiness of $\mathrm{L}(\mathbb{A})$ can be checked in time $\mathcal{O}(R(\mathtt{size}(\mathbb{A}))^{R'(\beta+D)})$ for some polynomials R and R'. The ExpTime upper bound of the nonemptiness problem for TCA is now a consequence of the above complexity expression.

▶ **Theorem 11.** *Nonemptiness problem for tree constraint automata is ExpTime-complete.*

4.4 Rabin Tree Constraint Automata

We can prove the ExpTime upper bound of the nonemptiness problem for Rabin TCA (Theorem 13) and follow the same lines of arguments as for TCA. Given a Rabin TCA $\mathbb{A} = (Q, \Sigma, D, \beta, Q_{\text{in}}, \delta, \mathcal{F})$, we define a Rabin tree automaton $A'_{\text{cons}(\mathbb{A})}$ such that $\text{L}(\mathbb{A}) \neq \varnothing$ iff there is $t \in \text{L}(A'_{\text{cons}(\mathbb{A})})$ that is satisfiable (cf. Lemma 4 for TCA). Moreover, we take advantage of $A_{\bigstar\text{C}}$ so that $\text{L}(\mathbb{A}) \neq \varnothing$ iff $\text{L}(A'_{\text{cons}(\mathbb{A})}) \cap \text{L}(A_{\bigstar\text{C}})$ is non-empty (cf. Lemma 8). It remains to determine the cost for testing nonemptiness of $\text{L}(A'_{\text{cons}(\mathbb{A})}) \cap \text{L}(A_{\bigstar\text{C}})$. Here is the counterpart of Lemma 9 (same kind of arguments).

▶ **Lemma 12.** *There is a Rabin tree automaton A s.t. $\text{L}(A) = \text{L}(A'_{cons(\mathbb{A})}) \cap \text{L}(A_{\bigstar C})$, the number of Rabin pairs is polynomial in $\beta + card(\mathcal{F})$, the number of locations is in $\mathcal{O}(card(\text{SatTypes}(\beta)) \times card(Q) \times 2^{P(\beta + card(\mathcal{F}))})$ for some polynomial $P(\cdot)$, and the transition relation can be decided in polynomial-time in $card(\delta) + \beta + card(\Sigma) + D + \texttt{MCS}(\mathbb{A})$.*

As for Lemma 10, we conclude that the nonemptiness problem for Rabin TCA can be solved in time $\mathcal{O}(R_1\big(\text{card}(Q) \times \text{card}(\delta) \times \texttt{MCS}(\mathbb{A}) \times \text{card}(\Sigma) \times R_2(\beta + \text{card}(\mathcal{F}))\big)^{R_2(\beta + \text{card}(\mathcal{F})) \times R_3(D)})$ for polynomials R_1, R_2 and R_3. The nonemptiness problem for Rabin TCA is also in ExpTime.

▶ **Theorem 13.** *The nonemptiness problem for Rabin TCA is ExpTime-complete.*

This result is mainly useful to characterize the complexity of SAT(CTL*(ℤ)) in Section 6.

5 Tree Constraint Automata for CTL(ℤ)

Below, we harvest the first results from what is achieved in the previous section: SAT(CTL(ℤ)) is in ExpTime. So, enriching the CTL models with numerical values interpreted in ℤ does not cause a complexity blow-up. We follow the automata-based approach and (after proving a refined version of the tree model property for CTL(ℤ)) the key step is to translate CTL(ℤ) formulae into equivalent TCA. Theorem 14 below is one of our main results.

▶ **Theorem 14.** *The satisfiability problem for* CTL(ℤ) *is ExpTime-complete.*

Sketch. ExpTime-hardness is inherited from CTL. For ExpTime-easiness, let ϕ be a CTL(ℤ) formula. A first step is to preprocess the formula into a formula in *simple form* (see definition in Section 2.2). Then, we can construct from a formula ϕ in simple form a TCA $\mathbb{A}_\phi$ s.t. ϕ is satisfiable iff $\text{L}(\mathbb{A}_\phi) \neq \varnothing$ and $\mathbb{A}_\phi$ satisfies the following properties.

- The degree D and the number of variables β are bounded by $\texttt{size}(\phi)$.
- The number of locations is bounded by $(D \times 2^{\texttt{size}(\phi)}) \times (\texttt{size}(\phi) + 1)$.
- The number of transitions is in $\mathcal{O}(2^{P(\texttt{size}(\phi))})$ for some polynomial $P(\cdot)$.
- The finite alphabet Σ in $\mathbb{A}_\phi$ is unary; $\texttt{MCS}(\mathbb{A}_\phi)$ is quadratic in $\texttt{size}(\phi)$.

By Lemma 10, the nonemptiness problem for TCA can be solved in time

$$\mathcal{O}(R_1\big(\text{card}(Q) \times \text{card}(\delta) \times \texttt{MCS}(\mathbb{A}) \times \text{card}(\Sigma) \times R_2(\beta)\big)^{R_2(\beta) \times R_3(D)}).$$

Since the transition relations of the automata $A_{\text{cons}(\mathbb{A})}$ and $A_{\bigstar\text{C}}$ can be built in polynomial-time, we get that nonemptiness of $\text{L}(\mathbb{A}_\phi)$ can be solved in exponential-time. ◀

Let ℕ be the concrete domain $(\mathbb{N}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{N}})$ for which we can also show that nonemptiness of TCA with constraints interpreted on ℕ has the same complexity as for TCA with constraints interpreted on ℤ. Let CTL(ℕ) be the variant of CTL(ℤ) with constraints interpreted on $\mathbb{N}$. As a corollary, SAT(CTL($\mathbb{N}$)) is ExpTime-complete. With the concrete domain $(\mathbb{Q}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{Q}})$, all the trees in $\mathrm{L}(A_{\mathrm{cons}(\mathbb{A})})$ are satisfiable (no need to intersect $A_{\mathrm{cons}(\mathbb{A})}$ with a hypothetical $A_{\star C}$, see e.g. [49, 4, 23, 38]), and therefore SAT(CTL($\mathbb{Q}$)) is in ExpTime too. TCA can be also used to show that the concept satisfiability w.r.t. general TBoxes for the description logic $\mathcal{ALCF}^{\mathcal{P}}(\mathbb{Z}_c)$ is in ExpTime [46, 45], see more details in [27, Section 5.2].

6 Complexity of the Satisfiability Problem for the Logic CTL*($\mathbb{Z}$)

We show that SAT(CTL*($\mathbb{Z}$)) can be solved in 2ExpTime. We follow the automata-based approach for CTL*, see e.g. [31, 30], but adapted to Rabin TCA. The main challenge here is to carefully check that essential steps for CTL* can be lifted to CTL*($\mathbb{Z}$), but also that computationally we are in a position to provide an optimal complexity upper bound.

Let us explain in short all steps necessary to obtain the result. We start by establishing a special form for CTL*($\mathbb{Z}$) formulae from which Rabin TCA will be defined, following ideas from [31] for CTL*. A CTL*($\mathbb{Z}$) state formula ϕ is in *special form* if it has the form below

$$\mathsf{E}\ (\mathtt{x} = 0)\ \wedge\ (\bigwedge_{i \in [1, D-1]} \mathsf{AGE}\ \Phi_i)\ \wedge\ (\bigwedge_{j \in [1, D']} \mathsf{A}\ \Phi'_j),$$

where the Φ_i's and the Φ'_j's are LTL($\mathbb{Z}$) formulae in simple form (see Section 2), for some $D \geqslant 1$, $D' \geqslant 0$. We can restrict ourselves to CTL*($\mathbb{Z}$) state formulae in special form (see the proof of [27, Proposition 6]).

▶ **Proposition 15.** *For every* CTL*($\mathbb{Z}$) *formula ϕ, one can construct in polynomial time in the size of ϕ a* CTL*($\mathbb{Z}$) *formula ϕ' in special form s.t. ϕ is satisfiable iff ϕ' is satisfiable.*

So ϕ' is also of polynomial size in the size of ϕ. Let us state a tree model property of special formulae, with a strict discipline on the witness paths. Proposition 16 below is a counterpart of [31, Theorem 3.2] but for CTL*($\mathbb{Z}$) instead of CTL*, see also the variant [38, Lemma 3.3].

▶ **Proposition 16.** *Let ϕ be a* CTL*($\mathbb{Z}$) *formula in special form built over $\mathtt{x}_1, \ldots, \mathtt{x}_\beta$. ϕ is satisfiable iff there is a tree $\mathfrak{t} : [0, D-1] \to \mathbb{Z}^\beta$ such that $\mathfrak{t}, \varepsilon \models \phi$ and for each $i \in [1, D-1]$, $\mathfrak{t}$ satisfies* AGE Φ_i *via i, that is, if $\mathfrak{t}, \mathbf{n} \models$* E Φ_i*, then Φ_i is satisfied on the path $\mathbf{n} \cdot i \cdot 0^\omega$.*

Proposition 16 justifies our restriction to infinite trees and to TCA in the rest of this section. Proposition 15 allows us to restrict our attention to constructing automata for formulae of (only) the form AGE Φ and A Φ, where Φ is a simple formula in LTL($\mathbb{Z}$). The first step is to translate simple formulae in LTL($\mathbb{Z}$) into equivalent *word* constraint automata (TCA with degree $D = 1$). Adapting the standard automata-based approach for LTL [69], we can show the following proposition (see the proof of [27, Proposition 8]).

▶ **Proposition 17.** *Let Φ be an* LTL($\mathbb{Z}$) *formula in simple form. There is a constraint word automaton $\mathbb{A}_\Phi$ such that $\{\mathfrak{w} : \mathbb{N} \to \mathbb{Z}^\beta \mid \mathfrak{w} \models \Phi\} = \mathrm{L}(\mathbb{A}_\Phi)$, and the following conditions hold.*

(I) *The number of locations in $\mathbb{A}_\Phi$ is bounded by $\mathtt{size}(\Phi) \times 2^{2 \times \mathtt{size}(\Phi)}$.*

(II) *The cardinality of δ in $\mathbb{A}_\Phi$ is in $\mathcal{O}(2^{P(\mathtt{size}(\Phi))})$ for some polynomial $P(\cdot)$.*

(III) *The maximal size of a constraint in $\mathbb{A}_\Phi$ is quadratic in $\mathtt{size}(\Phi)$.*

We can now construct, for every $i \in [0, D-1]$, a TCA $\mathbb{A}_i$ such that $\mathrm{L}(\mathbb{A}_i) = \{\mathfrak{t} : [0, D-1]^* \to \mathbb{Z}^\beta \mid \mathfrak{t} \models \mathsf{AGE}\ \Phi_i$ and $\mathfrak{t}$ satisfies AGE Φ_i via $i\}$. The idea is to construct $\mathbb{A}_i$ so that it starts off the word constraint automaton $\mathbb{A}_{\Phi_i}$ at each node $\mathbf{n}$ of the tree and runs it down the designated path $\mathbf{n} \cdot i\ \cdot 0^\omega$ to check whether Φ_i actually holds along this path. This can be easily done for AGE Φ_i; however, for formulas of the form A Φ'_j, for this construction to be correct, the underlying constraint word automaton $\mathbb{A}'_j$ must be *deterministic*, that is, for all locations s, letters $\mathtt{a}$ and pairs of valuations $(\boldsymbol{z}, \boldsymbol{z}') \in \mathbb{Z}^{2\beta}$, there exists in $\mathbb{A}'_j$ *at most* a single transition $(s, \mathtt{a}, \Theta, s')$ such that $\mathbb{Z} \models \Theta(\boldsymbol{z}, \boldsymbol{z}')$. A well-known construction to transform nondeterministic Büchi automata to equivalent deterministic Rabin automata is due to Safra [60, Theorem 1.1]. An important step towards the optimal complexity for CTL*($\mathbb{Z}$) is to show that it is possible to lift this construction to word constraint automata, which is a result of its own interest. A special attention is given to the cardinality of the transition relation and to the size of the constraints in transitions, as these two parameters are, a priori, unbounded in constraint automata but essential to perform a forthcoming complexity analysis.

▶ **Theorem 18.** *Let* $\mathbb{A} = (Q, \Sigma, \beta, Q_{in}, \delta, F)$ *be a Büchi word constraint automaton involving the constants* $\mathfrak{d}_1, \ldots, \mathfrak{d}_\alpha$*. There is a deterministic Rabin word constraint automaton* $\mathbb{A}' = (Q', \Sigma, \beta, Q'_{in}, \delta', \mathcal{F})$ *such that* $\mathrm{L}(\mathbb{A}) = \mathrm{L}(\mathbb{A}')$ *verifying the following quantitative properties.*

(I) *card*(Q') *is exponential in card*(Q) *and the number of Rabin pairs in* $\mathbb{A}'$ *is bounded by* $2 \cdot card(Q)$ *(same bounds as in [60, Theorem 1.1]).*

(II) *The constraints in the transitions are from* $\mathrm{SatTypes}(\beta)$*, are of size cubic in* $\beta + \max(\lceil log(|\mathfrak{d}_1|)\rceil, \lceil log(|\mathfrak{d}_\alpha|)\rceil)$ *and* $card(\delta') \leqslant card(Q')^2 \times card(\Sigma) \times ((\mathfrak{d}_\alpha - \mathfrak{d}_1) + 3)^{2\beta} \times 3^{\beta^2}$.

This and Proposition 17 lead us to the result below on LTL($\mathbb{Z}$) formulae in simple form.

▶ **Corollary 19.** *Let* Φ *be an* LTL($\mathbb{Z}$) *formula in simple form built over the variables* $\mathtt{x}_1, \ldots, \mathtt{x}_\beta$ *and the constants* $\mathfrak{d}_1, \ldots, \mathfrak{d}_\alpha$*. There exists a deterministic Rabin word constraint automaton* $\mathbb{A}_\Phi$ *such that* $\{\mathfrak{w} : \mathbb{N} \to \mathbb{Z}^\beta \mid \mathfrak{w} \models \Phi\} = \mathrm{L}(\mathbb{A}_\Phi)$*, and the following conditions hold.*

(I) *The number of locations in* $\mathbb{A}_\Phi$ *is bounded by* $2^{2^{P^\dagger(\mathtt{size}(\Phi))}}$ *for some polynomial* $P^\dagger(\cdot)$*.*

(II) *The number of Rabin pairs is bounded by* $2 \times \mathtt{size}(\Phi) \times 2^{2 \times \mathtt{size}(\Phi)}$.

(III) *The cardinality of* δ *in* $\mathbb{A}_\Phi$ *is bounded by card*$(\mathrm{SatTypes}(\beta)) \times 2^{2^{P^\dagger(\mathtt{size}(\Phi))+1}}$.

(IV) $\mathtt{MCS}(\mathbb{A}_\Phi)$ *is cubic in* $\beta + \max(\lceil log(|\mathfrak{d}_1|)\rceil, \lceil log(|\mathfrak{d}_\alpha|)\rceil)$*, i.e. polynomial in* $\mathtt{size}(\Phi)$*.*

This enables us to use the idea illustrated above for formulas of the form $\mathsf{AGE}\ \Phi_i$ also for formulas of the form $\mathsf{A}\ \Phi'_j$, and define Rabin TCA $\mathbb{A}'_j$ such that $\mathrm{L}(\mathbb{A}'_j) = \{\mathbbm{t} : [0, D-1]^* \to \mathbb{Z}^\beta \mid \mathbbm{t} \text{ satisfies } \mathsf{A}\ \Phi'_j\}$. We are now ready to perform the final step towards the main result of this section. Let us recapitulate what we have so far.

- One can define a TCA $\mathbb{A}_0$ with two locations such that $\mathrm{L}(\mathbb{A}_0)$ is the set of trees $\mathbbm{t} : [0, D-1]^* \to \mathbb{Z}^\beta$ such that $\mathbbm{t}(\varepsilon)(\mathtt{x}_1) = 0$, to handle $\mathsf{E}(\mathtt{x}_1 = 0)$ in formulae in special form.
- For all $1 \leqslant i < D$, there are (Büchi) TCA $\mathbb{A}_i$ such that $\mathrm{L}(\mathbb{A}_i)$ is the set of trees $\mathbbm{t} : [0, D-1]^* \to \mathbb{Z}^\beta$ such that $\mathbbm{t}, \varepsilon \models \mathsf{AGE}\ \Phi_i$ and $\mathbbm{t}$ satisfies $\mathsf{AGE}\ \Phi_i$ via i. Recall that TCA can be seen as Rabin TCA with a single Rabin pair.
- For all $1 \leqslant j \leqslant D'$, there are Rabin TCA $\mathbb{A}'_j$ such that $\mathrm{L}(\mathbb{A}'_j)$ is the set of trees $\mathbbm{t}$ such that $\mathbbm{t}$ satisfies $\mathsf{A}\ \Phi_j$, with an exponential number of Rabin pairs in $\mathtt{size}(\Phi)$.

To define a Rabin TCA $\mathbb{A}$ such that $\mathrm{L}(\mathbb{A}) = \mathrm{L}(\mathbb{A}_0) \bigcap_{i \in [1, D-1]} \mathrm{L}(\mathbb{A}_i) \bigcap_{j \in [1, D']} \mathrm{L}(\mathbb{A}'_j)$, and then use the complexity bounds previously established, we need the result below (see the full proof in [27, Section 6.5]).

▶ **Lemma 20.** *Let* $(\mathbb{A}_k)_{1 \leqslant k \leqslant n}$ *be a family of Rabin TCA such that* $\mathbb{A}_k = (Q_k, \Sigma, D, \beta, Q_{k,in}, \delta_k, \mathcal{F}_k)$*, card*$(\mathcal{F}_k) = N_k$ *and* $N = \prod_k N_k$*. There is a Rabin TCA* $\mathbb{A}$ *such that* $\mathrm{L}(\mathbb{A}) = \bigcap_k \mathrm{L}(\mathbb{A}_k)$ *and*

- *the number of Rabin pairs is equal to* N*;* $\mathtt{MCS}(\mathbb{A}) \leqslant n + \mathtt{MCS}(\mathbb{A}_1) + \cdots + \mathtt{MCS}(\mathbb{A}_n)$,
- *the number of locations (resp. transitions) is less than* $\big(\prod_k card(Q_k)\big)(2n)^N$ *(resp.* $\prod_k card(\delta_k)$*).*

Putting all results together, the nonemptiness of $\mathrm{L}(\mathbb{A})$ can be checked in double-exponential time in $\mathtt{size}(\phi)$, leading to Theorem 21 below, which is the main result of the paper. It answers open questions from [11, 15, 16, 46].

▶ **Theorem 21.** SAT(CTL*($\mathbb{Z}$)) *is 2ExpTime-complete.*

2ExpTime-hardness is from SAT(CTL*) [66, Theorem 5.2]. As a corollary, SAT(CTL*($\mathbb{N}$)) is also 2ExpTime-complete. Furthermore, assuming that $<_{\mathrm{pre}}$ is the prefix relation on $\{0,1\}^*$, we can use the reduction from [22, Section 4.2] to conclude SAT(CTL*($\{0,1\}^*, <_{\mathrm{pre}}$)) is 2ExpTime-complete too. Furthermore, as observed earlier, when the concrete domain is $(\mathbb{Q}, <, =, (=_{\mathfrak{d}})_{\mathfrak{d} \in \mathbb{Q}})$, all the trees in $\mathrm{L}(A_{\mathrm{cons}(\mathbb{A})})$ are satisfiable, and therefore SAT(CTL*($\mathbb{Q}$)) is also in 2ExpTime, which is already known from [38, Theorem 4.3].

7 Concluding Remarks

We developed an automata-based approach to solve SAT(CTL($\mathbb{Z}$)) and SAT(CTL*($\mathbb{Z}$)), by introducing tree constraint automata that accept infinite data trees with data domain $\mathbb{Z}$. The nonemptiness problem for tree constraint automata with Büchi acceptance conditions (resp. with Rabin pairs) is ExpTime-complete, see Theorem 11 (resp. Theorem 13). The difficult part consists in proving the ExpTime-easiness for which we show how to substantially adapt the material in [45, Section 5.2] that guided us to design the correctness proof of ($\bigstar^{\mathrm{C}}$). The work [46] was indeed a great inspiration but we adjusted a few statements from there (see also [27]). We recall that ($\bigstar$) in [46] is not fully correct (see Section 4.2) as we need to add constants (leading to the variant condition ($\bigstar^{\mathrm{C}}$)). Moreover, our construction of the automaton in Lemma 7 does depend on the number of variables unlike [46, Proposition 26]. This is crucial for complexity, as it is related to the number of Rabin pairs. We also use [30] more precisely than [46, p.621] as we handle non-binary trees. In short, we introduced TCA for which we characterise complexity of the non-emptiness problem (providing a few improvements to [46]). We left aside the question of the expressiveness of TCA, which is interesting but out of the scope of this paper.

This lead us to show that SAT(CTL($\mathbb{Z}$)) is ExpTime-complete (Theorem 14), and SAT(CTL*($\mathbb{Z}$)) is 2ExpTime-complete (Theorem 21). The only decidability proof for SAT(CTL*($\mathbb{Z}$)) done so far, see [15, Theorem 32], is by reduction to a decidable second-order logic. Our complexity characterisation for SAT(CTL*($\mathbb{Z}$)) provides an answer to several open problems related to CTL*($\mathbb{Z}$) fragments, see e.g. [11, 38, 15, 16, 46]. We believe that our results on TCA can help to establish complexity results for other logics (see also Section 6 about a domain for strings and [33, Section 4] to handle more concrete domains).

References

1 S. Abriola, D. Figueira, and S. Figueira. Logics of repeating values on data trees and branching counter systems. In *FoSSaCS'17*, volume 10203 of *LNCS*, pages 196–212, 2017.

2 E.G. Amparore, S. Donatelli, and F. Gallà. A CTL* model checker for Petri nets. In *Petri Nets'20*, volume 12152 of *LNCS*, pages 403–413. Springer, 2020.

3 F. Baader and J. Rydval. Using model theory to find decidable and tractable description logics with concrete domains. *JAR*, 66(3):357–407, 2022.

4 Ph. Balbiani and J.F. Condotta. Computational complexity of propositional linear temporal logics based on qualitative spatial or temporal reasoning. In *FroCoS'02*, volume 2309 of *LNAI*, pages 162–173. Springer, 2002.

5 K. Bartek and M. Lelyk. Modal mu-calculus with atoms. In *CSL'17*, pages 30:1–30:21. Leibniz-Zentrum für Informatik, LIPICS, 2017.

6 B. Bednarczyk and O. Fiuk. Presburger Büchi tree automata with applications to logics with expressive counting. In *WoLLIC'22*, volume 13468 of *LNCS*, pages 295–308. Springer, 2022.

7 A. Bhaskar and M. Praveen. Realizability problem for constraint LTL. In *TIME'22*, volume 247 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

8 M. Bojańczyk. A bounding quantifier. In *CSL'04*, volume 3210 of *LNCS*, pages 41–55. Springer, 2004.

9 M. Bojańczyk and Th. Colcombet. Bounds in ω-Regularity. In *LiCS'06*, pages 285–296. IEEE Computer Society, 2006.

10 M. Bojańczyk and S. Toruńczyk. Weak MSO+U over infinite trees. In *STACS'12*, LIPIcs, pages 648–660, 2012.

11 L. Bozzelli and R. Gascon. Branching-time temporal logic extended with Presburger constraints. In *LPAR'06*, volume 4246 of *LNCS*, pages 197–211. Springer, 2006.

12 L. Bozzelli and S. Pinchinat. Verification of gap-order constraint abstractions of counter systems. *TCS*, 523:1–36, 2014.

13 C. Carapelle. *On the satisfiability of temporal logics with concrete domains*. PhD thesis, Leipzig University, 2015.

14 C. Carapelle, A. Kartzow, and M. Lohrey. Satisfiability of CTL* with constraints. In *CONCUR'13*, LNCS, pages 455–463. Springer, 2013.

15 C. Carapelle, A. Kartzow, and M. Lohrey. Satisfiability of ECTL* with constraints. *Journal of Computer and System Sciences*, 82(5):826–855, 2016.

16 C. Carapelle and A.-Y. Turhan. Description Logics Reasoning w.r.t. General TBoxes is Decidable for Concrete Domains with the EHD-property. In *ECAI'16*, volume 285, pages 1440–1448. IOS Press, 2016.

17 K. Čerāns. Deciding properties of integral relational automata. In *ICALP'94*, volume 820 of *LNCS*, pages 35–46. Springer, 1994.

18 A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *JACM*, 28(1):114–133, 1981.

19 R. Condurache, C. Dima, Y. Oualhdaj, and N. Troquard. Rational Synthesis in the Commons with Careless and Careful Agents. In *AAMAS'21*, pages 368–376, 2021.

20 B. Cook, H. Khlaaf, and N. Piterman. On automation of CTL* verification for infinite-state systems. In *CAV'15*, volume 9206 of *LNCS*, pages 13–29. Springer, 2015.

21 N. Decker, P. Habermehl, M. Leucker, and D. Thoma. Ordered navigation on multi-attributed data words. In *CONCUR'14*, volume 8704 of *LNCS*, pages 497–511, 2014.

22 S. Demri and M. Deters. Temporal logics on strings with prefix relation. *Journal of Logic and Computation*, 26:989–1017, 2016.

23 S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *I & C*, 205(3):380–415, 2007.

24 S. Demri and R. Gascon. Verification of qualitative $\mathbb{Z}$ constraints. *TCS*, 409(1):24–40, 2008.

25 S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM ToCL*, 10(3), 2009.

26 S. Demri and K. Quaas. Concrete domains in logics: a survey. *ACM SIGLOG News*, 8(3):6–29, 2021.

27 S. Demri and K. Quaas. Constraint automata on infinite data trees: From CTL(Z)/CTL*(Z) to decision procedures. CoRR, abs/2302.05327, 2023. `arXiv:2302.05327`.

28 A. Deutsch, R. Hull, and V. Vianu. Automatic verification of database-centric system. *SIGMOD Record*, 43(3):5–17, 2014.

29 E.A. Emerson and J. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *JACM*, 33:151–178, 1986.

30 E.A. Emerson and C.S. Jutla. The complexity of tree automata and logics of programs. *SIAM Journal of Computing*, 29(1):132–158, 2000.

31 E.A. Emerson and P. Sistla. Deciding full branching time logic. *Information and Control*, 61:175–201, 1984.

32 L. Exibard, E. Filiot, and A. Khalimov. Church synthesis on register automata over linearly ordered data domains. In *STACS'21*, volume 187 of *LIPIcs*, pages 28:1–28:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

33 L. Exibard, E. Filiot, and A. Khalimov. A generic solution to register-bounded synthesis with an application to discrete orders. In *ICALP'22*, volume 229 of *LIPIcs*, pages 122:1–122:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

34 P. Felli, M. Montali, and S. Winkler. CTL* model checking for data-aware dynamic systems with arithmetic. In *IJCAR'22*, volume 13385 of *LNCS*, pages 36–56. Springer, 2022.

35 P. Felli, M. Montali, and S. Winkler. Linear-time verification of data-aware dynamic systems with arithmetic. In *AAAI'22*, pages 5642–5650. AAAI Press, 2022.

36 D. Figueira. *Reasoning on words and trees with data*. PhD thesis, ENS Cachan, 2010.

37 D. Figueira. Decidability of Downward XPath. *ACM ToCL*, 13(4):1–40, 2012.

38 R. Gascon. An automata-based approach for CTL* with constraints. *Electronic Notes in Theoretical Computer Science*, 239:193–211, 2009.

39 S. Göller, Ch. Haase, J. Ouaknine, and J. Worrell. Branching-time model checking of parametric one-counter automata. In *FoSSaCS'12*, volume 7213 of *LNCS*, pages 406–420. Springer, 2012.

40 J.F. Groote and R. Mastescu. Verification of temporal properties of processes in a setting with data. In *AMAST'98*, volume 1548 of *LNCS*, pages 74–90, 1998.

41 R. Iosif and X. Xu. Alternating automata modulo first order theories. In *CAV'19*, volume 11562 of *LNCS*, pages 46–63, 2019.

42 M. Jurdziński and R. Lazić. Alternating automata on data trees and XPath satisfiability. *ACM ToCL*, 12(3):19:1–19:21, 2011.

43 A. Kartzow and Th. Weidner. Model checking constraint LTL over trees. *CoRR*, abs/1504.06105, 2015. `arXiv:1504.06105`.

44 O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *JACM*, 47(2):312–360, 2000.

45 N. Labai. *Automata-based reasoning for decidable logics with data values*. PhD thesis, TU Wien, May 2021.

46 N. Labai, M. Ortiz, and M. Simkus. An Exptime Upper Bound for $\mathcal{ALC}$ with integers. In *KR'20*, pages 425–436. Morgan Kaufman, 2020.

47 S. Lasota and I. Walukiewicz. Alternating timed automata. *ACM ToCL*, 9(2):10:1–10:27, 2008.

48 A. Lechner, R. Mayr, J. Ouaknine, A. Pouly, and J. Worrell. Model checking flat freeze LTL on one-counter automata. *Logical Methods in Computer Science*, 14(4), 2018.

49 C. Lutz. Interval-based temporal reasoning with general TBoxes. In *IJCAI'01*, pages 89–94. Morgan-Kaufmann, 2001.

50 C. Lutz. *The Complexity of Description Logics with Concrete Domains*. PhD thesis, RWTH, Aachen, 2002.

51 C. Lutz. Description logics with concrete domains – A survey. In *Advances in Modal Logics Volume 4*, pages 265–296. King's College Publications, 2003.

52 C. Lutz. NEXPTIME-complete description logics with concrete domains. *ACM ToCL*, 5(4):669–705, 2004.

53 C. Lutz and M. Milicić. A Tableau Algorithm for Description Logics with Concrete Domains and General Tboxes. *JAR*, 38(1-3):227–259, 2007.

54 R. Mayr and P. Totzke. Branching-time model checking gap-order constraint systems. *Fundamenta Informaticae*, 143(3–4):339–353, 2016.

55 D. Muller and E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. *TCS*, 141(1–2):69–107, 1995.

56 F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM ToCL*, 5(3):403–435, 2004.

57 D. Peteler and K. Quaas. Deciding Emptiness for Constraint Automata on Strings with the Prefix and Suffix Order. In *MFCS'22*, volume 241 of *LIPIcs*, pages 76:1–76:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

58 M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the AMS*, 141:1–35, 1969.

59 P. Revesz. *Introduction to Constraint Databases.* Springer, New York, 2002.

60 S. Safra. *Complexity of Automata on Infinite Objects.* PhD thesis, The Weizmann Institute of Science, Rehovot, 1989.

61 L. Segoufin and S. Toruńczyk. Automata based verification over linearly ordered data domains. In *STACS'11*, pages 81–92, 2011.

62 H. Seidl, Th. Schwentick, and A. Muscholl. Counting in trees. In *Logic and Automata: History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 575–612. Amsterdam University Press, 2008.

63 F. Song and Z. Wu. On temporal logics with data variable quantifications: decidability and complexity. *I & C*, 251:104–139, 2016.

64 W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B, Formal models and semantics*, pages 133–191. Elsevier, 1990.

65 Sz. Torunczyk and Th. Zeume. Register automata with extrema constraints, and an application to two-variable logic. *Logical Methods in Computer Science*, 18(1), 2022.

66 M. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *STOC'85*, pages 240–251. ACM, 1985.

67 M. Vardi and Th. Wilke. Automata: from logics to algorithms. In *Logic and Automata: History and Perspectives*, number 2 in Texts in Logic and Games, pages 629–736. Amsterdam University Press, 2008.

68 M. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

69 M. Vardi and P. Wolper. Reasoning about infinite computations. *I & C*, 115:1–37, 1994.

70 S. Vester. On the complexity of model-checking branching and alternating-time temporal logics in one-counter systems. In *ATVA'15*, volume 9364 of *LNCS*, pages 361–377. Springer, 2015.

71 Th. Weidner. *Probabilistic Logic, Probabilistic Regular Expressions, and Constraint Temporal Logic.* PhD thesis, University of Leipzig, 2016.

Visibility and Separability for a Declarative Linearizability Proof of the Timestamped Stack

Jesús Domínguez
IMDEA Software Institute, Madrid, Spain
Universidad Politécnica de Madrid, Spain

Aleksandar Nanevski
IMDEA Software Institute, Madrid, Spain

Abstract

Linearizability is a standard correctness criterion for concurrent algorithms, typically proved by establishing the algorithms' linearization points (LP). However, LPs often hinder abstraction, and for some algorithms such as the timestamped stack, it is unclear how to even identify their LPs. In this paper, we show how to develop declarative proofs of linearizability by foregoing LPs and instead employing axiomatization of so-called visibility relations. While visibility relations have been considered before for the timestamped stack, our study is the first to show how to derive the axiomatization systematically and intuitively from the sequential specification of the stack. In addition to the visibility relation, a novel *separability* relation emerges to generalize real-time precedence of procedure invocation. The visibility and separability relations have natural definitions for the timestamped stack, and enable a novel proof that reduces the algorithm to a simplified form where the timestamps are generated atomically.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases Linearizability, Visibility Relations, Timestamped Stack

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.30

Related Version *Extended Version*: https://arxiv.org/abs/2307.04720 [5]

Funding This work was partially supported by the ERC project MATHADOR (ERC2016-COG-724464), the Madrid regional government and EU project BLOQUES (S2018/TCS-4339), and the Spanish MCIN and EU project PRODIGY (TED2021-132464B-I00).

1 Introduction

A concurrent data structure is *linearizable* [11] if in every concurrent execution history of the structure's exportable methods, the method invocations can be ordered linearly just by permuting *overlapping* invocations, so that the obtained history is *sequentially sound*; that is, executing the methods sequentially in the linear order produces the same outputs that the methods had in the concurrent history. In other words, every concurrent history is equivalent to a sequential one where methods execute without interference, i.e., *atomically.*

While linearizability is a standard correctness criterion, proving that sophisticated data structures are linearizable is far from trivial. The most common approach is to first describe the linearization points (LPs) of the methods that the data structure exports. Given an execution of a method (henceforth, event), its LP is the moment at which the event's effect can be considered to have occurred abstractly, in the sense that the linearization order of the events is determined by the real-time order of the chosen LPs. LPs are described operationally by indicating the line in the code together with a run-time condition under which the line applies. The proof then proceeds by a simulation argument to show that the effect of the invocation abstractly occurs at the declared line.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 30; pp. 30:1–30:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

While LPs lead to a complete proof method [16], the operational nature of the LP description leads to very low-level proofs. Sometimes, it may even be unclear how to describe the position of the LPs in the first place. An alternative, more *declarative* approach, that offers higher levels of abstraction, has been proposed by Henzinger et al. [10]. It advocates foregoing LPs in favor of axiomatizing how the events of the structure depend on each other. Such dependence relation has since been termed *visibility relation* in the literature [17], and has been used to axiomatize concurrent queues [10], stacks [3, 7], and snapshot algorithms [14]. In these cases, the higher abstraction capabilities of visibility relations (compared to LPs) enabled that linearizability proofs of different implementations of a data structure can share significant proof components, or that a linearizability proof can be developed in the first place where an LP-based proof did not exist. Nevertheless, despite these recent successes, developing visibility-based proofs remains an undeveloped area, with every proof approaching the axiomatization in its own manner, without any specific systematization.

This paper advances the visibility approach by proposing that the axiomatization of concurrent structures should rely on a *separability* relation between events, in addition to the visibility relation. Separability relation partially characterizes when two events are *abstractly* non-overlapping, with one event logically preceding the other. Thus, it is the abstract counterpart to the "returns-before" relation, which is standard in the literature, and holds between two events if, in real time, the first event terminates before the second begins.

We employ the visibility and separability relations in tandem to derive a new axiomatization and linearizability proof for the concurrent structure of the *timestamped stack*, initially designed and proved linearizable by Dodds et al. [3] and Haas [7]. Since its inception, the timestamped stack has achieved some notoriety for the difficulty of its linearizability proof, as it has so far resisted an operational description of its LPs, and simulation-based proof attempts. For example, Khyzha et al. [12] verified the timestamped *queue* by a simulation-based approach, but did not scale to the stack. Bouajjani et al. [1] employed forward simulation on a simplified variant of the stack where timestamps are allocated atomically, but did not attempt the general variant, where the timestamp allocation is a more complex non-atomic operation that produces behaviors not observed in the atomic case. The original proof by Dodds et al. is a large case analysis that mixes visibility relations with *approximate* LP descriptions, and then adapts and corrects both as the proof advances. However, the axioms and the definitions of the visibility relations have been justified only technically, and have remained unconnected to the intuition behind the structure's design.

By axiomatizing both separability and visibility relations, we derive the following contributions: (1) We obtain a linearizability proof that elides LPs, and is thus more declarative than the proof of Dodds et al.; (2) The proof's declarative nature allows us to first consider the simpler variant of the algorithm with atomic timestamp allocation, and then show that the general variant reduces to the atomic case. The staged proof is more intuitive than if we attempted the general case directly, which is what Dodds et al. do.; (3) Our contribution goes beyond a new proof for the timestamped stack, as it suggests a *systematic* way to axiomatize concurrent data structures in the visibility style. More specifically, we show that the visibility relation naturally emerges when one transforms an obvious state-based sequential axiomatization of stacks to the concurrent setting with histories. In the process, the separability relation also naturally emerges, because one is immediately forced to generalize the returns-before relation. So obtained axioms identify the abstractions that are essential for understanding the algorithm, and strongly guide the remaining proof. Finally, our approach to axiomatization applies to other concurrent algorithms as well, and we comment in Section 5 how we did so for RDCSS and MCAS of Harris et al. [8] and some other structures. Of course, the generality of the approach remains to be evaluated on a wider set of examples.

```
 1: $pools$ : NODE[$maxThreads$];
 2: $TS$, $ID$ : INT = 0;
 3:
 4: proc $push$ ($v$ : VAL)
 5:     NODE $n$ =
 6:         NODE{$v, \infty, pools[TID], false, ID{+}{+}$ };
 7:     $pools[TID] = n$;
 8:     STAMP $ts = newTimestamp()$;
 9:     $n.stamp = ts$;
10:
11: proc $newTimestamp$ ()
12:     INT $ts_1 = TS$;
13:     $pause()$;
14:     INT $ts_2 = TS$;
15:     if $ts_1 \neq ts_2$ then
16:         return $[ts_1, ts_2 - 1]$;
17:     else if $CAS(TS, ts_1, ts_1 + 1)$ then
18:         return $[ts_1, ts_1]$;
19:     else
20:         return $[ts_1, TS - 1]$;
```

```
21: record NODE:
22:     $val$ : VAL; $stamp$ : STAMP; $next$ : NODE;
23:     $taken$ : BOOL; $id$ : INT ;
24:
25: proc $pop$ ()
26:     BOOL $suc = false$; NODE $chosen$;
27:     while not $suc$ do
28:         STAMP $maxT = -\infty$;
29:         $chosen = null$;
30:         for $i$ from $0$ to $maxThreads - 1$ do
31:             NODE $n = pools[i]$;
32:             while $n.taken$ and $n.next \neq n$ do
33:                 $n = n.next$;
34:             STAMP $ts = n.stamp$;
35:             if $maxT <_{\mathrm{T}} ts$ then
36:                 $chosen = n$; $maxT = ts$;
37:         if $chosen \neq null$ then
38:             $suc = CAS(chosen.taken, false, true)$;
39:     return $chosen.val$;
```

Figure 1 Pseudocode of a simplified TS-stack.

2 The Timestamped Stack and its Timestamps

The timestamped stack (TS-stack) keeps an array of *pools*, indexed by thread IDs; one pool for each thread. A pool is a linked list of nodes. The array index identifying the pool (line 1 in Figure 1) stores the head node of the pool list, and each node (lines 22-23) stores a value *val*, timestamp *stamp*, the *next* node in the list, a boolean *taken* indicating if the value has been taken by some pop, and a unique identifier *id* for the node.[1] Each thread can only insert values in its own pool by allocating a node at the head of the list. A value is logically removed from the pool once its *taken* flag is set to *true*.[2]

The *push* procedure inserts a new node containing the pushed value v into the pool of the executing thread with thread id *TID*. More specifically, in line 6, *push* allocates a new node with the value v, infinite timestamp, *next* pointing to the current head of the pool, taken flag set to *false*, and fresh unique identifier, where *ID*++ denotes an atomic fetch and increment on the global counter *ID*. Then, the new node is set as the new head of the *TID* pool (line 7), a new timestamp is generated (line 8) and assigned to the node (line 9) as a replacement for the original infinity timestamp. We will discuss infinity timestamps and the *newTimestamp* procedure further below.

The *pop* procedure traverses the pools (loop at line 30), searching for an untaken node with a maximal timestamp in the partial order $<_{\mathrm{T}}$, updating the current maximum in the variable *maxT* (lines 35-36). Once a maximal node is found, *pop* attempts to remove it by CAS-ing on its *taken* flag at line 38. The *pop* procedure restarts (loop at line 27) if it was not able to take a maximal node at line 38.

The role of $<_{\mathrm{T}}$ is to endow TS-stack with a LIFO discipline whereby an element with a larger timestamp (i.e, the more recently pushed element), is popped first. In the concurrent setting, however, the meaning of "more recent" is not as straightforward as in the sequential setting, as the definition of linearizability allows that overlapping operations can be linearized

[1] Unique identifiers *id* are ghost code (gray color in Figure 1), introduced solely for use in proofs.

[2] For presentation purposes, we simplified the original algorithm, but treat a more general form in Appendix B.2 [5]. The two versions exhibit the same challenges, and use the sames axiomatization and definitions of visibility and separability. The differences between them are discussed in Section 5.

in either order. In particular, if two invocations of *push* overlapped, they can actually be popped in either order. To reflect this property of linearizability, the order $<_T$ is *partial* as opposed to total. However, to be sequentially sound, it is of essence that if two pushes did *not* overlap, then the more recent push is indeed popped first.[3] This is why the implementation of *newTimestamp* should satisfy the property that two non-overlapping calls to *newTimestamp* produce timestamps that actually *are* ordered by $<_T$.

There are several ways in which one can implement *newTimestamp* to satisfy this property, and Figure 1 shows the particularly efficient variant proposed by Dodds et al. [3]. We will return to this variant promptly. However, for purposes of understanding and proving the algorithm linearizable, one may consider a simpler version whereby timestamps are integers, and *newTimestamp* is implemented to keep a global counter that is *atomically* fetch-and-incremented on each call, returning the current count as the fresh timestamp. Such an atomic implementation results in $<_T$ that is actually a total order, and much simpler to analyze than the efficient variant in Figure 1. We will use the atomic implementation as a stepping stone in our proof; we will prove it linearizable first, and then show that the linearizability argument for the efficient variant *reduces* to the atomic case.

The reason to consider a non-atomic implementation at all is that the atomic one suffers from a performance issue that threads contend on the global timestamp counter. The efficient variant from Figure 1 improves on this by introducing *interval* timestamps of the form $[a, b]$ for integers $a \leq b$, where $[a, b] <_T [c, d]$ holds if $b < c$ in the standard integer order. Obviously, so defined $<_T$ is only a partial order, as it does not order *every* two interval timestamps. Nevertheless, it still suffices for linearizability, because if two push events are assigned overlapping interval timestamps, such events must overlap as well, and thus do not constrain the order in which they are popped in a linearization.

The *newTimestamp* from Figure 1 still keeps a global counter *TS*, as the atomic variant would, but it does not always synchronize accesses to it. In particular, *TS* is first read twice into ts_1 and ts_2 (lines 12 and 14, respectively). In the common case when some thread interfered on *TS* (i.e., $ts_1 \neq ts_2$), the method generates an interval with endpoint $ts_2 - 1$, and terminates without having performed any synchronization. Some synchronization is required only when no interference is detected (i.e., $ts_1 = ts_2$). In that case, *newTimestamp* *CAS*-es over *TS* (line 17), to atomically increment *TS*. As *CAS* is an expensive operation, invoking *pause*() in line 13 increases the probability of interference, and thus decreases the need for *CAS*. If the *CAS* succeeds, an interval with endpoint ts_1 is returned. If the *CAS* fails, some other thread increased *TS*, and the method returns an endpoint $TS - 1$. In all cases, when *newTimestamp* terminates, *TS* has been increased either by the executing thread or by another thread, and the generated interval's endpoint is strictly smaller than the current value of *TS*. Thus, a subsequent non-overlapping invocation of *newTimestamp* will produce an interval that is strictly larger in $<_T$. This ensures that two sequentially non-overlapping pushes generate non-overlapping interval timestamps.

Note that *newTimestamp* could return the *same* interval timestamp for two different overlapping invocations. For example, with initial $TS = 0$, a thread T_1, after reading TS the first time at line 12 (returning 0), waits at line 13 while another thread T_2 fully executes *newTimestamp*, meaning that T_2 increased TS at line 17 and returned timestamp $[0, 0]$. When T_1 resumes, it again reads TS at line 14 (returning 1), and so returns $[0, 0]$.

[3] Further assuming that the pops also did not overlap among themselves or with the pushes.

(A_1) Non-empty *pop* $\quad v :: S \xrightarrow{pop()\ \langle v\rangle} S$

(A_2) Empty *pop* $\quad [\,] \xrightarrow{pop()\ \langle \text{EMPTY}\rangle} [\,]$

(A_3) *push* $\quad S \xrightarrow{push(v)\ \langle tt\rangle} v :: S$

(a) State-based sequential specification.

(B_1) LIFO

$$u_1 \lessdot o_1 \wedge u_1 \sqsubset u_2 \sqsubset o_1 \implies \exists o_2.\ u_2 \lessdot o_2 \wedge o_2 \sqsubset o_1$$

(B_2) Pop uniqueness

$$u \lessdot o_1 \wedge u \lessdot o_2 \implies o_1 = o_2$$

(B_3) Dependences occur in the past

$$u \lessdot o \implies u \sqsubset o$$

$(B_{4.1})$ Non-empty *pop*

$$o = pop()\,\langle v\rangle \wedge v \neq \text{EMPTY} \implies \exists u.\ u \lessdot o \wedge v = u.in$$

$(B_{4.2})$ Empty *pop*

$$o_1 = pop()\,\langle \text{EMPTY}\rangle \implies \forall u.\ u \sqsubset o_1 \implies \exists o_2.\ u \lessdot o_2 \wedge o_2 \sqsubset o_1$$

$(B_{4.3})$ *push*

$$u = push(\_)\,\langle v\rangle \implies v = tt$$

(b) History-based sequential specification. Relation $\lessdot : \text{Ev} \times \text{Ev}$ is abstract, and $u.in$ is event u's input.

Figure 2 State-based and history-based sequential specifications for stacks. Variables u, o, and their indexed variants, range over pushes and pops, respectively.

Finally, $<_\text{T}$ is formally augmented with infinite timestamps $-\infty$ and ∞, so that $-\infty <_\text{T} t <_\text{T} \infty$ for any timestamp t generated by *newTimestamp*. This enables *pop* to start its search with minimum timestamp $-\infty$ (line 28). Similarly, *push* can assign maximum timestamp ∞ to a fresh node (line 6) before assigning it a finite timestamp; an intervening pop could take such a fresh node immediately, as the node is the most recent.

3 Axiomatizing Visibility and Separability

3.1 Sequential History Specifications and Visibility Relations

Following Henzinger et al. [10], we start the development of visibility relations by introducing *history-based specifications* for our data structure. History-based specifications describe relationships between the data structure's procedures in an execution history. They are significantly different from the perhaps more customary state-based specifications that describe the actions of a procedure in terms of input and output state. However, history-based specifications scale better to the concurrent setting, which is why concurrent consistency criteria such as linearizability are invariably defined in terms of execution histories.

In this section we focus on *sequential* histories in order to introduce the idea of *visibility relation* in a simple way, before generalizing to concurrent histories in Section 3.2. A sequential history is a sequence of the form $[proc(in_1)\langle out_1\rangle, \ldots, proc(in_n)\langle out_n\rangle]$, where $proc(in_i)\langle out_i\rangle$ means that $proc(in_i)$ executed *atomically* and produced output out_i. We term *event* each element in a sequential history h, and Ev denotes the set of all events in h.

Figure 2 illustrates the distinction between sequential state-based and history-based specifications for stacks. For the state-based specification in Figure 2a, let us denote by $S \xrightarrow{proc(in)\ \langle out\rangle} S'$ the statement that event *proc* with input *in* executes atomically on stack S, produces output *out* and modifies the stack into S'. Axiom A_1 says that a *pop* removes the top element v from a non-empty stack and returns v. Axiom A_2 says that *pop* returns EMPTY when the stack is empty, leaving the stack unchanged. Axiom A_3 says $push(v)$ inserts v into the stack as the new top element, returning the trivial value tt.

Figure 2b shows the history-based sequential specification for stacks. The specification utilizes the *visibility relation* $\lessdot$ to capture a push-pop causal dependence between events. In particular, $u \lessdot o$ means that "event o pops a value that event u pushed onto the stack". We usually say that u is *visible* to o, or that o *observes* u. Under this interpretation, axioms $B_1, ..., B_{4.3}$ state the following expected properties.[4]

Axiom B_1 (LIFO) states that more recent pushes are popped first. More specifically, if o_1 observes u_1 (i.e, $u_1 \lessdot o_1$) and u_2 is a later push executing between u_1 and o_1 (i.e., $u_1 \sqsubset u_2 \sqsubset o_1$), then u_2 must be popped before o_1 pops u_1, otherwise the value pushed by u_1 would not be at the top of the stack for o_1 to take. Relation $\sqsubset$ is the *returns-before* relation (with $\sqsubseteq$ its reflexive closure), where $x \sqsubset y$ means that x terminated before y started. Note that $\sqsubset$ is a total order on events, as in a sequential execution, different events cannot overlap.

Axiom B_2 (Pop uniqueness) says that a push is observed by at most one pop.

Axiom B_3 (Dependences occur in the past) says that if a pop depends on a push, then the push executes before the pop.

Axioms $B_{4.1}$ (Non-empty *pop*), $B_{4.2}$ (Empty *pop*), and $B_{4.3}$ essentially are the counterparts of the state-based sequential axioms A_1-A_3, respectively, as we show next.

Axiom $B_{4.1}$ says that a *pop*() $\langle v \rangle$ event o observes a push u that pushed v. This axiom, along with B_1-B_3, ensures that o relates to u as in the following diagram.

$$\begin{tikzcd}
S \arrow[r, "u"] & v :: S \arrow[r, "u_i"] & \dots \arrow[r, "o_j"] & v :: S \arrow[r, "u_k"] & \dots \arrow[r, "o_l"] & v :: S \arrow[r, "o=pop()\langle v \rangle"] & S
\arrow[from=1-1, to=1-7, dash, dashed, bend left=20, "\lessdot"]
\arrow[from=1-2, to=1-4, dash, dashed, bend left, "\lessdot"]
\arrow[from=1-4, to=1-6, dash, dashed, bend left, "\lessdot"]
\end{tikzcd}$$

In particular: (i) u executes before o (by axiom B_3, because $u \lessdot o$), (ii) every push between u and o is popped before o (by axiom B_1), and each push is popped exactly once (by axiom B_2). Thus, once o executes, the value v pushed by the observed u, is actually the most recent unpopped value, i.e., it is on the top of the stack. Subsequent pops cannot observe this value anymore either (again by axioms B_1 and B_2), thus the stack is modified from $v :: S$ to S. This explains that $B_{4.1}$ is essentially a history-based version of A_1.

Similarly, axiom $B_{4.2}$ states that if a *pop*() $\langle$EMPTY$\rangle$ event o occurs, then every push before o must have been popped before o, as this ensures that the stack is empty when o is reached. Hence, the axiom is counterpart to A_2.

Finally, axiom $B_{4.3}$ says that the output of a push event is the trivial value *tt*. The axiom imposes no conditions on the stack, as a value can always be pushed. In this, $B_{4.3}$ is the counterpart to A_3 which also imposes no conditions on the input stack, and posits that push's output value is trivial. However, unlike A_3, $B_{4.3}$ does not directly says that the value is pushed on the top of the stack, as that aspect is captured by the relationships between pushes and pops described by $B_{4.1}$.

3.2 Concurrent Specifications and Separability Relations

Concurrent execution histories do not satisfy the sequential axioms in Figure 2b for two related reasons. First, concurrent events can *overlap* in real time. As a consequence, the axioms B_1 (LIFO), B_3 (Dependencies occur in the past), and $B_{4.2}$ (Empty *pop*) are too

[4] Our paper will make heavy use of several different relations. To help the reader keep track of them, we denote the relations by symbols that graphically associate to the relation's meaning. For example, we use $\lessdot$ for the visibility relation, because the symbol graphically resembles an eye.

(C_1) Concurrent LIFO

$$u_1 \lessdot o_1 \wedge o_1 \not\underline{\ltimes}\, u_2 \not\underline{\ltimes}\, u_1 \implies \exists o_2.\ u_2 \lessdot o_2 \wedge o_2 \ltimes o_1$$

(C_2) Pop uniqueness

$$u \lessdot o_1 \wedge u \lessdot o_2 \implies o_1 = o_2$$

(C_3) No future dependences

$$x \prec^+ y \implies y \not\sqsubseteq x$$

(C_4) Return value completion

$$\exists v.\ \mathcal{Q}_{x,v} \wedge (x \in T \implies v = x.out)$$

(a) Concurrent specification. Relations $\lessdot, \ltimes : \text{Ev} \times \text{Ev}$ are abstract.

Constraint relation

$$\prec\ \mathrel{\widehat{=}}\ \lessdot \cup \ltimes$$

Returns-before relation

$$e_1 \sqsubset e_2 \mathrel{\widehat{=}} e_1.end <_{\mathbb{N}} e_2.start$$

Set of terminated events

$$T \mathrel{\widehat{=}} \{e \mid e.end \neq \bot\}$$

Closure of terminated events

$$\overline{T} \mathrel{\widehat{=}} \{e \mid \exists t \in T.\ e \prec^* t\}$$

$$\mathcal{Q}_{o_1,v} \mathrel{\widehat{=}} \begin{cases} \exists u.\ u \lessdot o_1 \wedge v = u.in & \text{if } v \neq \text{EMPTY} \\ \forall u.\ o_1 \not\underline{\ltimes}\, u \implies \exists o_2.\ u \lessdot o_2 \wedge o_2 \ltimes o_1 & \text{if } v = \text{EMPTY} \end{cases} \qquad \mathcal{Q}_{u,v} \mathrel{\widehat{=}} v = tt$$

(b) Defined notions.

Figure 3 Concurrent history-based specification for stacks. Variables u, o, and their indexed variations, range over pushes and pops in $\overline{T}$, respectively. Variables x, y range over $\overline{T}$. Variable e, and its indexed variations, range over Ev. $x.out$ denotes x's output.

restrictive, as they force events to be non-overlapping (i.e., disjoint in time) due to the use of the returns-before relation $\sqsubset$. Second, events can no longer be treated as atomic; thus event's start and end times (if the event terminated) must be taken into account. As a consequence, axioms $B_{4.1}$, $B_{4.2}$, and $B_{4.3}$ must be modified to account for the output of an unfinished event not being available yet. We continue using Ev for the set of events in the concurrent history. We denote by $e.start$ and $e.end$ the start and end time of event e, respectively; for example, for the implementation in Figure 1, a *push* event starts when line 5 executes, and ends when line 9 executes. We use the standard order relation on natural numbers $<_{\mathbb{N}}$ to compare start and end times.

Figure 3 shows the modified axioms that address the above issues. Importantly, in addition to the visibility relation, the axioms utilize the *separability relation* $x \ltimes y$ to capture that "event x is separable before y", i.e., x should be linearized before y.[5] The reason for the separation depends on the particular stack implementation, but is kept abstract in the axioms. Correspondingly, the relation $\ltimes$ is also kept abstract. We now explain how the concurrent axioms in Figure 3 are *systematically* obtained from the sequential ones in Figure 2b.

Axiom C_1 is obtained from B_1 by replacing $\sqsubset$ with $\ltimes$ or with the (negation of the) reflexive closure $\underline{\ltimes}$, following the rules below. The goal is to relax the real-time strong separation imposed by $\sqsubset$ with a more permissive separation of $\ltimes$.

- If subformula $a \sqsubset b$ occurs in a condition of an implication (negative occurrence), it is replaced with $b \not\underline{\ltimes}\, a$. Notice the flip in the arguments and the negation.
- If subformula $a \sqsubset b$ occurs in the conclusion of an implication (positive occurrence), it is replaced with $a \ltimes b$.

These rules have the following justification. Let us suppose we have a formula $\phi \mathrel{\widehat{=}} a \sqsubset b \implies c \sqsubset d$ in some sequential axiom. In the sequential case, $\sqsubset$ is a total order, which means that ϕ is equivalent to $b \sqsubseteq a \vee c \sqsubset d$. After directly replacing $\sqsubset$ for $\ltimes$, we obtain

[5] The symbol $\ltimes$ twists $\sqsubset$, suggesting that $\ltimes$ relaxes (i.e., is a twist on) returns-before relation $\sqsubset$.

$b \mathrel{\underline{\ltimes}} a \vee c \ltimes d$, which is further equivalent to $\psi \mathrel{\widehat{=}} b \mathrel{\not\underline{\ltimes}} a \implies c \ltimes d$. Comparing ϕ and ψ, we see that ψ's condition is flipped, replaced, and negated, while its conclusion is only replaced. An important aspect of our procedure is that negative occurrences of $\ltimes$ in ψ are themselves negated. Thus, intuitively, ψ as a whole remains positive with respect to $\ltimes$. Positive formulas remain true under extensions of $\ltimes$, which is crucial, as the linearizability proof will involve extending $\ltimes$ until reaching a total order.

Axiom C_2 is unchanged compared to B_2.

Axiom C_3 is obtained from B_3 as follows. In the sequential specification, $\lessdot$ was the only relation encoding dependences between events, but now we have two relations encoding dependences, $\lessdot$ and $\ltimes$. To collect them, we define a new relation $\prec \mathrel{\widehat{=}} \lessdot \cup \ltimes$ which we call *constraint* relation.[6] We can consider modifying Axiom B_3 into $x \prec y \implies x \sqsubset y$ to say that any dependence x of y must terminate before y starts. However, such a modification of B_3 is too stringent, as it does not allow x to overlap with y. Instead, we relax the conclusion to say that an event cannot depend on itself or events from the future, i.e., $x \prec y \implies y \not\sqsubseteq x$. Finally, we get axiom C_3 by replacing $\prec$ with its transitive closure $\prec^+$ to account for *indirect* dependences of y; e.g., in $x_1 \prec x_2 \prec y$, event x_1 is an indirect dependence of y. Hence, C_3 reads "any direct or indirect dependence does not execute in the future, and events do not depend on themselves".

To understand Axiom C_4, we need to consider the set T of all *terminated* events and its closure under the constraint relation $\overline{T} \mathrel{\widehat{=}} \{e \in \mathrm{Ev} \mid \exists t \in T.\ e \prec^* t\}$. As usual, $\prec^*$ is the reflexive-transitive closure of $\prec$. The reason for considering this set is that the variable x over which the axiom implicitly quantifies ranges over $\overline{T}$.

It is standard in linearizability that the linearization order contains all the terminated events, plus selected unterminated events with fictitious, but suitable, outputs. The selected unterminated events are typically those that executed their effect, which then influenced others, and must thus be included for sequential soundness. The set $\overline{T}$ precisely determines the events to be included by saturating the set of terminated events T under $\prec$.

Axiom C_4 then codifies when an output v is suitable for an event x by means of the *postcondition predicate* $\mathcal{Q}_{x,v}$. In particular, C_4 says that v exists such that $\mathcal{Q}_{x,v}$. If $x \in \overline{T}$ is unterminated, we use that v as the fictitious output. If x is terminated ($x \in T$), then v must be x's actual output. The postcondition predicate $\mathcal{Q}_{x,v}$ describes how x and v relate in the case of stacks. It is obtained by coalescing the axioms $B_{4.1}$, $B_{4.2}$ and $B_{4.3}$, which themselves describe the outputs of stack events in the sequential setting, and which we first modify according to the systematic transformation outlined above.

We henceforth call the axioms in Figure 3, *visibility-style axioms.* These axioms imply linearizability of any stack implementation satisfying them,

▶ **Theorem 3.1.** *Let D be an arbitrary implementation of a concurrent stack. If there are relations $\ltimes$ and $\lessdot$ definable using D such that the visibility-style axioms hold, then D is linearizable.*

The proof starts with the relation $\lhd \mathrel{\widehat{=}} (\prec \cup \sqsubset)^+$, i.e, the transitive closure of the union of $\prec$ and $\sqsubset$. Then, it shows that $\lhd$ is a partial order that can be extended to a sequentially sound total order $\leq$ by using the visibility-style axioms. Since $\leq$ contains $\prec$, this means that relations $\lessdot$ and $\ltimes$ define ordering constraints that linearization respects.

[6] The symbol $\prec$ is like an eye with no iris; thus, "blinder" than $\lessdot$, reflecting that $\prec$ is a superset of $\lessdot$.

4 Visibility and Separability for the TS-stack

By Theorem 3.1, to prove linearizability for the TS-stack, it suffices to define the relations $\lessdot$ and $\ltimes$ and show that they satisfy the axioms from Figure 3. We carry out this proof in two stages: In Section 4.2 we prove linearizability when the *newTimestamp* procedure is implemented by an atomic fetch-and-increment operation on a global counter *TS*, as discussed in Section 2. In Section 4.3 we show how the general case of interval timestamps reduces to the atomic case. The lifting exploits that the difference between the atomic and interval cases is only in the implementation of *newTimestamp*.

For simplicity, in both cases we explicitly exclude *elimination pairs* from the discussion. An elimination pair consists of a push and an overlapping pop event that takes the value pushed. The elision allows the discussion to only consider pushes with *finite* timestamps. Indeed, every push is first assigned an infinite timestamp (line 6 in Figure 1), which is then refined into a finite one in line 9. If a push u, having not yet reached line 9, is taken by some pop o, then u and o overlap, and hence form an elimination pair.[7]

Also, in both cases, we utilize the abstraction we call *spans*, to define the visibility and separability relations. A span of an event is the interval in which the event accesses the shared state of the stack. We could trivially take the span to be the whole interval of the event, but in the case of TS-stack we can tighten it as discussed below. In this sense, a span is a generalization of LPs; being an interval, rather than a single point, it *approximates* where the LP of an event lies, but allows for some uncertainty as to the LPs exact position.

The span of the *push* procedure starts when the new node is linked as the first node of the pool (line 7), as this is the moment when the new node becomes available for other events to see. The span ends when a finite timestamp is assigned to the new node (line 9). Notice how the span encompasses all the commands of *push* that change the pool or the new node.

The span of the *pop* procedure starts at the infinity stamp assignment (line 28) of the last iteration of the pools scan. The span ends at the successful CAS at line 38 which takes the node for the pop to return. Again, the span covers all the commands of *pop* that change the pools or the taken node. These are all included in the last iteration of the pools scan, because in all the prior iterations, the CAS modifying the pools must have failed.

We formalize spans as pairs of rep events (a, b), where a and b are the initial and final rep event in the span, respectively. We denote by *start* (b) and *end* (b) the standard projection functions for span b. Rep events are generated by the invocation of a code line inside a procedure. For example, invoking line 7 in Figure 1 produces a rep event. The set of all rep events in an execution history is denoted as Rep. The distinction between events (Ev) and rep events is standard in linearizability [11]. We denote by $<$ the real-time order between rep events. We consider only fully-formed spans; for example, if *pop* has not executed its successful CAS, then it has no span.

We also extend our notion of timestamp into *abstract timestamp*. An abstract timestamp is a pair (i, t), where i is the (ghost) id of a node, and t is a (plain) timestamp. The extension is motivated by the observation explained in Section 2 that two pushes may actually generate

[7] Eliding elimination pairs when dealing with stacks is justified because such pairs can be linearized simply as a push that is immediately followed by a pop. The idea was originated by Hendler et al. [9] and was also employed in Haas' PhD dissertation [7], though with a different motivation from us and with a different soundness proof. For example, to prove the elimination sound, Haas shows how elimination pairs could be put back into the histories from which they have been removed. In contrast, we define visibility and separability relations that exclude elimination pairs, and show in Appendix B.1 [5], how to extend the relations iteratively, one elimination pair at a time. The extension adds some bulk, but does not change the structure of the proof that we illustrate in this section.

the same (plain) timestamp. By attaching the node id i to the timestamp t, we differentiate such cases. We use "timestamp" to refer to abstract timestamps or plain timestamps when the adjective can be inferred from the context.

We also utilize the following notation.

- Given event e, $\mathcal{S}\ e$ is the unique span executed by e. The function is undefined if the argument event has not completed its span.
- Given spans a, b, the relation $a \sqsubset^S b$ means that a finished before b started. $\sqsubseteq^S$ denotes its reflexive closure.
- For a push u, $\mathit{id}\ u$ is the unique id of the node that u inserted into the pool in line 7. Similarly, for a pop o, $\mathit{id}\ o$ is the unique id of the node that o took at the successful CAS in line 38. If u and o have not executed the mentioned lines, id is undefined.
- For a push u, $t_s\ u$ is the abstract timestamp $(\mathit{id}\ u, t)$, combining $\mathit{id}\ u$ with the timestamp t that u assigned at line 9. In particular, t is always *finite*, because *newTimestamp* only generates finite timestamps. Similarly, for a pop o, $t_s\ o$ is the abstract timestamp $(\mathit{id}\ o, t)$, combining $\mathit{id}\ o$ with the timestamp t of the taken node that o read in line 34. Generally, $t_s\ o$ may return an infinite plain timestamp; however, if elimination pairs are excluded, then timestamps are finite, as explained before. If an event x has not executed its span, $t_s\ x$ is undefined.
- Abstract timestamps admit the following partial order defined out of $<_{\mathrm{T}}$ on plain timestamps, where we overload the symbol $<_{\mathrm{T}}$ without confusion.

  $$(i_1, t_1) <_{\mathrm{T}} (i_2, t_2) \ \mathrel{\widehat{=}}\ t_1 <_{\mathrm{T}} t_2$$

- We define when push u and pop o form an elimination pair.

  $$u \ \textsc{Elim}\ o \ \mathrel{\widehat{=}}\ \mathit{id}\ u = \mathit{id}\ o \wedge u \not\sqsubset o$$

  In English: (1) o pops the node that u pushed ($\mathit{id}\ u = \mathit{id}\ o$), and (2) u and o overlap. Events u and o overlap if $u \not\sqsubset o$ and $o \not\sqsubset u$, but it is not necessary to explicitly check $o \not\sqsubset u$, as that follows from $\mathit{id}\ u = \mathit{id}\ o$ and a structural invariant that o cannot pop a node that has not been pushed yet (Appendix B.1 [5]).
- The set of events that occur in elimination pairs is $E \mathrel{\widehat{=}} \{x \mid \exists y.\ x \ \textsc{Elim}\ y \vee y \ \textsc{Elim}\ x\}$. As we explicitly exclude elimination pairs from the presentation, we assume that each event variable x occurring in the forthcoming definitions is such that $x \notin E$. In Section 5 we comment how elimination pairs are placed back into consideration.

4.1 Key Abstractions and Invariants

When pop misses a push. The key for understanding TS-stacks is explaining what it means for a pop o to have *missed* a push u. Informally, a miss occurs when o, in its scan of the pools, takes a push u' with a smaller timestamp than that of u (hence, u' is less recent than u). This is critical, because o taking a less recent push than available is seemingly a violation of the LIFO order. However, this does not actually have to be so in the case of TS-stacks, where, for example, it is fine for u to insert into the pool after o has already scanned past the point of insertion. We can say that u occurred too late to really be available for o to pop, and we simply linearize u after o. The following definition formalizes when o misses u (i.e., when u occurs too late for o), focusing on the atomic timestamp case.

$$\begin{aligned} \textsc{Miss}\ o\ u \ \mathrel{\widehat{=}}\ & t_s\ o \ <_{\mathrm{T}}\ t_s\ u \ \wedge \\ & \forall o'.\ t_s\ u = t_s\ o' \implies \mathit{end}\,(\mathcal{S}\ o) < \mathit{end}\,(\mathcal{S}\ o') \end{aligned} \tag{1}$$

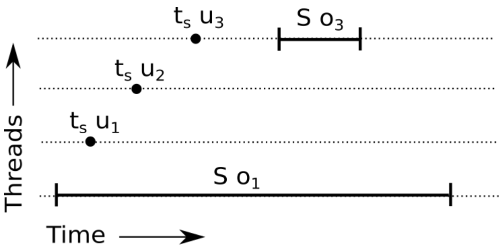


Figure 4 Possible execution showing three atomic timestamp generation rep events for push events u_1, u_2, u_3, labeled by their generated timestamps; and two pop spans for pop events o_1 and o_3. Spans are shown as line segments and rep events (being atomic) as dots. The timestamps are strictly increasing $t_s\ u_1 <_{\mathrm{T}} t_s\ u_2 <_{\mathrm{T}} t_s\ u_3$. Event o_1 took u_1, while o_3 took u_3. The events will be linearized as u_1, o_1, u_2, u_3, o_3. In particular, o_1 must be linearized before u_2.

The first conjunct directly says that for o to miss u, it must be that o takes a node with a smaller timestamp than that of u. The second conjunct adds that, intuitively, u *remains untaken* during the execution of o. Indeed, if u is taken by o' ($t_s\ u = t_s\ o'$), the definition requires that the span of o' finishes after the span of o ($\mathit{end}\,(\mathcal{S}\ o) < \mathit{end}\,(\mathcal{S}\ o')$). That is, the CAS that sets the *taken* flag in the node of u executes after the span of o. In other words, if u is taken at all, then it is taken *after* the span of o.

Figure 4 shows a push u_2 that overlaps with a pop o_1, but o_1 takes a push u_1 whose timestamp is smaller than that of u_2. In our definition, $\textsc{Miss}\ o_1\ u_2$ holds because u_2 remains untaken on the stack after o_1 terminates. $\textsc{Miss}\ o_1\ u_2$ indicates that we must linearize o_1 before u_2. And indeed, this is consistent with the situation in the figure, as any order where u_2 appears before o_1 violates some linearizability requirement. For example, the order u_1, u_2, o_1 is sequentially unsound because o_1 pops u_1 while u_2 is the top of the stack, while u_2, u_1, o_1 does not respect the ordering of the timestamps of u_1 and u_2.[8]

Continuing with Figure 4, o_1 *does not miss* u_3, even though u_3 also overlaps with o_1, and o_1 takes u_1 whose timestamp is smaller than that of u_3. In our definition, $\neg\textsc{Miss}\ o_1\ u_3$ because the span of o_3 ends before the span of o_1. $\neg\textsc{Miss}\ o_1\ u_3$ indicates no restrictions on the ordering between o_1 and u_3. For example, the only linearization order of Figure 4 is u_1, o_1, u_2, u_3, o_3, but this is forced by the existence of u_2. Removing u_2, the orders u_1, u_3, o_3, o_1 (where u_3 appears before o_1) and u_1, o_1, u_3, o_3 (where u_3 appears after o_1) are both valid.

Misses start late. Having defined $\textsc{Miss}\ o\ u$, we can now explain the most important invariants of the TS-stack, again focused on the atomic timestamps. The first invariant says that a push u missed by a pop o has a span that starts *after* the pop's span starts. In other words, a missed push starts after the pop that missed it.

$$\textsc{Miss}\ o\ u \implies \mathit{start}\,(\mathcal{S}\ o) < \mathit{start}\,(\mathcal{S}\ u) \tag{2}$$

To intuit why (2) is an invariant, consider a situation when o misses u but u's span starts before o's. In that case, u's pool contains u's node *before* o even starts its scan. Thus o's scan will encounter u and proceed to either take u, or take an even more recent push. At any rate, o will not take a push with a timestamp below that of u; thus, $\neg(\textsc{Miss}\ o\ u)$.

[8] We linearize pushes by the order of their timestamps.

Disjoint pushes order timestamps. The next invariant is that pushes with disjoint spans, produce ordered timestamps. Intuitively, this is so because disjoint push spans make disjoint calls to *newTimestamp*, which in turn generate ordered timestamps as explained in Section 2.

$$\mathcal{S}\ u_1 \sqsubset^S \mathcal{S}\ u_2 \implies t_s\ u_1 <_{\mathrm{T}} t_s\ u_2 \tag{3}$$

4.2 Case: Atomic Timestamps

We next define the visibility $\lessdot$ and separability $\ltimes$ relations for atomic timestamps.

$$u \lessdot o \mathrel{\widehat{=}} t_s\ u = t_s\ o \tag{4}$$

$$u_1 \ltimes u_2 \mathrel{\widehat{=}} t_s\ u_1\ <_{\mathrm{T}}\ t_s\ u_2 \tag{5}$$

$$o \ltimes u \mathrel{\widehat{=}} \exists u'.\ \textsc{Miss}\ o\ u'\ \wedge\ t_s\ u'\ \leq_{\mathrm{T}}\ t_s\ u \tag{6}$$

$$o_2 \ltimes o_1 \mathrel{\widehat{=}} t_s\ o_1\ <_{\mathrm{T}}\ t_s\ o_2\ \wedge\ \neg\exists u'.\ \textsc{Miss}\ o_1\ u'\ \wedge\ t_s\ u'\ \leq_{\mathrm{T}}\ t_s\ o_2 \tag{7}$$

The definition of $\lessdot$ relates u and o if they have the same timestamp (i.e., o took u).

The definition of $\ltimes$ comes with three clauses, motivated by the form of the axioms from Figure 3. In particular, we need to separate a push from a push ($u_1 \ltimes u_2$), a pop from a push ($o \ltimes u$), and a pop from a pop ($o_2 \ltimes o_1$), but not a push from a pop, as only the first three clauses of $\ltimes$ appear in the axioms.

The clause $u_1 \ltimes u_2$ naturally orders push events according to their timestamps.

The clause $o \ltimes u$ extends $\textsc{Miss}\ o\ u'$ to account for pushes being ordered by their timestamps, as per the previous clause. It says that pop o is separated before push u, if there is a push u' that was missed by o, and the timestamp of u' is below (or equals) that of u. For example, in Figure 4 we have $o_1 \ltimes u_2$ and $o_1 \ltimes u_3$.

The clause $o_2 \ltimes o_1$ separates pops inversely to the order of the taken timestamps, or equivalently, inversely to the order of the taken pushes, but under the condition that o_1 *did not miss any push with a timestamp below* o_2. The last requirement is important. For example, if we ignored it in Figure 4, we would obtain $o_3 \ltimes o_1$ since $t_s\ u_1 <_{\mathrm{T}} t_s\ u_3$. But this order is sequentially unsound; the pushes being ordered as u_1, u_2, u_3, after o_3 takes u_3, the value pushed by u_2 is at the top of the stack. But then o_1 cannot execute next, as we need an intervening pop to remove u_2.

It is worth mentioning that we arrived at the definition of the clause $o_2 \ltimes o_1$ by formal symbol manipulation aimed at fulfilling axiom C_1 (Concurrent LIFO) after the definitions of the other clauses have been unfolded in C_1. In hindsight, this may have been expected, as the clauses $u_1 \ltimes u_2$ and $o \ltimes u$ are hypotheses of C_1, while $o_2 \ltimes o_1$ is in the conclusion.

The engineering of the (uniquely determined) definition of the clause $o_2 \ltimes o_1$ thus makes the proof of axiom C_1 out of definitions (4)-(7) quite straightforward, but for one important observation. Because the axiom contains negations of several clauses of $\ltimes$, unfolding the definitions of these clauses reveals comparisons of the form $t_s\ x \not<_{\mathrm{T}} t_s\ y$, where the relation $<_{\mathrm{T}}$ appears negated. The proof then crucially relies on $<_{\mathrm{T}}$ being total, so that we can *flip* the negated comparisons into the form $t_s\ y\ \leq_{\mathrm{T}}\ t_s\ x$. It is the requirement of totality of $<_{\mathrm{T}}$ that makes the described development specific to atomic timestamps. In Section 4.3, we shall see how to adapt to interval timestamps where $<_{\mathrm{T}}$ is *not* total.

▶ **Theorem 4.1.** *Given* $\lessdot$ *and* $\ltimes$ *as in* (4)–(7)*, the TS-stack with atomic timestamps satisfies the invariants in Section 4.1 and the axioms in Figure 3, and is thus linearizable by Theorem 3.1.*

The characteristic part of the proof is showing that the axiom C_3 (no future dependences) holds, which is where we rely on the invariants (2) and (3). This proof generates obligations, one of which is that $o \ltimes u \sqsubset o$ for some push u and pop o is impossible, as such u depends on o which is in u's future. The proof proceeds by contradiction: suppose $o \ltimes u \sqsubset o$. By definition of $o \ltimes u$, there exists a push u' missed by o such that $t_s\ u' \leq_{\mathrm{T}} t_s\ u$. By invariant (2), u' starts after o starts, and since $u \sqsubset o$, it must also be $u \sqsubset u'$. But then, by invariant (3), it is also $t_s\ u <_{\mathrm{T}} t_s\ u'$. In other words, $t_s\ u <_{\mathrm{T}} t_s\ u' \leq_{\mathrm{T}} t_s\ u$, a contradiction.

4.3 Case: Interval Timestamps

The proof from Section 4.2 does not directly apply to the interval timestamps because proving axiom C_1 (Concurrent LIFO) relies on the totality of $<_{\mathrm{T}}$ in order to flip the negated inequalities $t_s\ x \not<_{\mathrm{T}} t_s\ y$ into positive facts $t_s\ y\ \leq_{\mathrm{T}}\ t_s\ x$. The relation $<_{\mathrm{T}}$ is total in the atomic case, but not in the interval case.

The key observation that allows us to recover the argument is that *whenever $<_{\mathrm{T}}$ is used to compare the timestamps of two push events in the proofs of the atomic case, at least one of the push events is invariably popped.* In other words, the proof does not actually require totality, but only the following weaker property of *pop-totality*. Formally, if R is a partial order on abstract timestamps, then R is pop-total if:

$$\begin{aligned}&\forall u_1\ u_2\ o.\, (t_s\ u_1 = t_s\ o) \vee (t_s\ u_2 = t_s\ o) \implies \\ &\qquad\qquad (t_s\ u_1)\ R\ (t_s\ u_2) \vee (t_s\ u_2)\ R\ (t_s\ u_1) \vee (t_s\ u_1 = t_s\ u_2)\end{aligned} \tag{8}$$

In English: if at least one of the pushes is taken, then the timestamps generated by the pushes are totally comparable under R.

As an illustration why the weaker property suffices, consider the hypotheses of the axiom C_1: these are $u_1 \lessdot o_1$, $o_1 \not\underline{\ltimes}\ u_2$ and $u_2 \not\underline{\ltimes}\ u_1$. Let us further assume that $<_{\mathrm{T}}$ in all the definitions is replaced by an arbitrary pop-total R. A common pattern throughout the proof of Theorem 4.1 is that three conjuncts of the above form appear together. Such combination entails that u_1 and u_2 are both popped, thus allowing us to flip any negated relation R in which $t_s\ u_1$ or $t_s\ u_2$ may appear.

Indeed, that u_1 is popped, and by o_1, follows from $u_1 \lessdot o_1$, which is defined as $t_s\ o_1 = t_s\ u_1$. To see that u_2 must also be popped consider the following. First, note that $o_1 \not\underline{\ltimes}\ u_2$ implies $\neg\textsc{Miss}\ o_1\ u_2$, by an easy derivation. Pushing the negation inside the definition of $\textsc{Miss}$ and substituting $t_s\ o_1 = t_s\ u_1$ derives $\neg(t_s\ u_1)\ R\ (t_s\ u_2) \vee \exists o'.\, t_s\ u_2 = t_s\ o' \wedge \ldots$. The second disjunct directly says that u_2 is popped by some o'. By pop-totality of R, the first disjunct implies $(t_s\ u_2)\ R\ (t_s\ u_1) \vee (t_s\ u_2) = (t_s\ u_1)$, and thus $t_s\ u_2 = t_s\ u_1$, because $\neg(t_s\ u_2)\ R\ (t_s\ u_1)$ by $u_2 \not\underline{\ltimes}\ u_1$. Thus, u_1 and u_2 are the same push, and u_2 is popped as well.

It follows that we could replicate the atomic case proof to the interval case, if we could replace $<_{\mathrm{T}}$ with some pop-total relation over *interval timestamps* throughout the definitions and proofs in Sections 4.1 and 4.2. We next define such a relation $\ll$ that includes $<_{\mathrm{T}}$.

$$\begin{aligned}t_2 \ll t_1\ &\mathrel{\widehat{=}}\ t_2 <_{\mathrm{T}} t_1\ \vee\ \exists u_1, u_2.\ t_s\ u_1 \not<_{\mathrm{T}} t_s\ u_2\ \wedge\ \textsc{Tb}\ u_1\ u_2\ \wedge \\ &\qquad\qquad\qquad\qquad\qquad t_2 \leq_{\mathrm{T}} t_s\ u_2\ \wedge\ t_s\ u_1 \leq_{\mathrm{T}} t_1 \\ \textsc{Tb}\ u_1\ u_2\ &\mathrel{\widehat{=}}\ \exists o_1.\ t_s\ u_1 = t_s\ o_1\ \wedge\ \forall o_2.\ t_s\ u_2 = t_s\ o_2 \implies \mathit{end}\,(\mathcal{S}\ o_1) < \mathit{end}\,(\mathcal{S}\ o_2)\end{aligned}$$

The key insight of the definition is that if two pushes u_1 and u_2 are not already ordered by $<_{\mathrm{T}}$, i.e., $t_s\ u_1 \not<_{\mathrm{T}} t_s\ u_2$, we could order their timestamps in $\ll$ in the order in which the pushes are popped. Indeed, if u_1 is *taken before* u_2 ($\textsc{Tb}\ u_1\ u_2$), then LIFO warrants that u_2

is linearized before u_1. We thus order u_2's timestamp before u_1's timestamp in $\ll$. It follows that $u_2 \ltimes u_1$ (assuming $\ll$ substitutes $<_{\mathrm{T}}$ in the definition of $u_2 \ltimes u_1$), and consequently that u_2 is linearized before u_1. The definition of $\ll$ further saturates the relation to include any $t_2 \ll t_1$ where $t_2 \leq_{\mathrm{T}} t_s\ u_2$ and $t_s\ u_1 \leq_{\mathrm{T}} t_1$, as then $t_2 \ll t_1$ is forced by $t_s\ u_2 \ll t_s\ u_1$.

Returning to taken-before, we define $\textsc{Tb}\ u_1\ u_2$ to hold of two pops u_1 and u_2 if: (1) u_1 is taken and u_2 is not, or (2) both are taken by pops o_1 and o_2, respectively. In the case (2) we require that the span of o_1 ends before the span of o_2, i.e., o_1 took its push before o_2 did.

One can now proceed to prove that $\ll$ is a strict partial order that is pop-total, that the invariants “Misses start late” and “Disjoint pushes order timestamps” from Section 4.1, continue to hold for the TS-stack with interval timestamps, after substituting $<_{\mathrm{T}} := \ll$ in the definition of $\textsc{Miss}$ (1), and definitions (2), (3) of the invariants. The visibility and separability relations for the TS-stack with interval timestamps are exactly as in (4)-(7) but with substitution $<_{\mathrm{T}} := \ll$, and our final theorem about the correctness of TS-stack is obtained simply by retracing the proof of Theorem 4.1.

▶ **Theorem 4.2.** *Let $\lessdot$ and $\ltimes$ defined as in (4)–(7) but under the substitution $<_{\mathrm{T}} := \ll$. The TS-stack with interval timestamps satisfies the invariants in Section 4.1 under the substitution, and the axioms in Figure 3, and is thus linearizable by Theorem 3.1.*

5 Discussion, Related and Future Work

Dealing with elimination pairs. To handle elimination pairs that were excluded in Section 4, we recursively define indexed families of visibility and separability relations, where $\lessdot_i$ and $\ltimes_i$ means that the first i elimination pairs have been added (Appendix B.1 [5]). At level 0, $\lessdot_0$ and $\ltimes_0$ are the relations from Section 4. At some limit level n, where n is the number of elimination pairs, we have the final relations $\lessdot_n$ and $\ltimes_n$ that consider all the events.

The theorems in Section 4 show that the visibility-style axioms in Figure 3 hold for events in $\overline{T} \setminus E$, i.e., $\overline{T}$ without elimination pairs. They are the base case of our proof in Appendix B.1 [5], which proceeds to inductively show that if the visibility-style axioms hold for the first i pairs, they continue to hold when the pair $i + 1$ is added.

Differences with the original algorithm. Figure 1 is a simplified version of the algorithm from Appendix B.2 [5]. The latter further treats elimination pair detection and node unlinking (i.e., node deallocation from memory). We consider the simplified version solely for presentation reasons, as the simplification still presents the same verification challenges and suffices to motivate the visibility and separability relations in Section 4. The definitions of these relations transfer to Appendix B.1 [5], where they serve as a basis for defining a family of augmented relations that deal with elimination pairs, as described above.

Having said this, the program that we treat in Appendix B.2 [5] still differs in a relatively minor way from the original program of Dodds et al. [3] in that we elide empty stack detection (i.e. pops returning EMPTY). This can be treated separately as an extra independent step in the proof [7], which means that considering empty pops changes neither the analysis we already presented in Section 4 nor the proof for elimination pairs in Appendix B.1.3 [5]. Nevertheless, we plan to augment the proof with an extra step that considers empty pops.

Related proofs. Dodds et al. [3] proof is also based on a visibility relation (their **val**), in addition to several other relations. However, our two axiomatizations and proofs differ significantly. Our axiomatization arises from a systematic transformation of a state-based sequential specification of stacks into a history-based concurrent specification, while that of Dodds et al. does not seem to derive from such prior principles, though it does suffice for the linearizability proof. The different axiomatizations give rise to different relations on histories as well. For example, their insert-remove (**ir**) relation is defined in terms of LPs of submodules. The objective in using LPs of submodules is to start with a definition for that may have linearizability violations, which then gets adjusted along the proof to remove such violations. In contrast, the definitions of our relations in Section 4 require no adjustments since they already lead to a correct linearization, albeit by eliding LPs. As a result, our relations are quite a bit more direct, and support better proof decomposition. In particular, our proof transfers from the easier atomic timestamp case to the more difficult interval timestamp case, whereas Dodds et al. immediately consider the interval case.

Bouajjani et al. [1] employs forward simulation on the atomic timestamp variant of the TS-stack, but do not attempt the interval timestamp variant. Our proof (Appendix B.2.2 [5]) does not employ simulations, and also lifts the atomic timestamp case to the interval timestamp case. The lifting exploits that the difference between the atomic and interval timestamp cases is not in the program structure, but only in the implementation of *newTimestamp*.

Visibility relations in other contexts. Our approach uses visibility and separability relations to model ordering dependencies between events. A general survey of the use of visibility relations in concurrency and distributed systems is given by Viotti and Vukolić [17]. Visibility relations and declarative proofs have also been utilized to specify consistency criteria weaker than linearizability (Emmi and Enea [6]), to introduce a specification framework for weak memory models (Raad et al. [15]), and to specify the RC11 memory model (Lahav et al. [13]).

In contrast to the above papers that focus on the semantics of consistency criteria, our use of visibility relations focuses on verifying specific algorithms and data structures, and is thus closer to the following work where visibility relations are applied to concurrent queues (Henzinger et al. [10, 2]), concurrent stacks (Dodds et al. [3] and Haas [7]), and memory snapshot algorithms (Öhman and Nanevski [14]). We differ from these in the addressed structures, or in the case of Dodds et al. in the structure of the proof and its components.

Our key innovation compared to these works is the introduction of the separability relation and its utilization to systematically axiomatize the stack structure in a novel way.

Visibility and separability as a general methodology. The pattern suggested by Sections 3.1 and 3.2, whereby one transforms a history-based sequential specification into a concurrent specification, by replacing the returns-before relation $\sqsubset$ with a separability relation $\ltimes$, points towards a general methodology for axiomatizing concurrent structures.

To test the generality of the approach, we have applied it – successfully ([4] and Appendix C.1 [5]) – to the RDCSS and MCAS algorithms of Harris et al. [8]. These algorithms write *descriptors* (a record with information about the task that a thread requires help with) into pointers, so that a thread that reads a descriptor can provide help by executing the described task. These algorithms implicitly "bunch" their help requests into related groups, and the separability relation models gaps between such bunches. On the other hand, the visibility relation models a writer-reader dependency, similarly to the push-pop dependency in this paper. We have also applied the approach to queues, where it derived a mildly streamlined variant of the queue axioms of Henzinger et al. [10, 2], and to locks, including readers-writers locks. In the future, we plan to study if this pattern applies to other concurrent data structures (e.g., memory snapshots, trees, lists, sets, etc.).

References

1 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *Computer Aided Verification (CAV)*, pages 542–563, 2017. doi:10.1007/978-3-319-63390-9_28.

2 Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. *Logical Methods in Computer Science (LMCS)*, 11(1), 2015. doi:10.2168/LMCS-11(1:20)2015.

3 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 2015. doi:10.1145/2676726.2676963.

4 Jesús Domínguez and Aleksandar Nanevski. Declarative linearizability proofs for descriptor-based concurrent helping algorithms. arXiv:2307.04653.

5 Jesús Domínguez and Aleksandar Nanevski. Visibility and separability for a declarative linearizability proof of the timestamped stack: Extended version. arXiv:2307.04720.

6 Michael Emmi and Constantin Enea. Weak-consistency specification via visibility relaxation. *Proc. ACM Program. Lang.*, 3(POPL):60:1–60:28, 2019. doi:10.1145/3290373.

7 Andreas Haas. *Fast Concurrent Data Structures Through Timestamping*. PhD thesis, University of Salzburg, 2015. URL: https://www.cs.uni-salzburg.at/~ahaas/papers/thesis.pdf.

8 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing (DISC)*, pages 265–279, 2002. doi:10.1007/3-540-36108-1_18.

9 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 206–215, 2004. doi:10.1145/1007912.1007944.

10 Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *International Conference on Concurrency Theory (CONCUR)*, pages 242–256, 2013. doi:10.1007/978-3-642-40184-8_18.

11 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

12 Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. Proving linearizability using partial orders. In *European Symposium on Programming (ESOP)*, pages 639–667, 2017. doi:10.1007/978-3-662-54434-1_24.

13 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 618–632, 2017. doi:10.1145/3062341.3062352.

14 Joakim Öhman and Aleksandar Nanevski. Visibility reasoning for concurrent snapshot algorithms. *Proc. ACM Program. Lang.*, 6(POPL):33:1–33:30, 2022. doi:10.1145/3498694.

15 Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL), 2019. doi:10.1145/3290381.

16 Gerhard Schellhorn, John Derrick, and Heike Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Logic*, 15(4), 2014. doi:10.1145/2629496.

17 Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, 2016. doi:10.1145/2926965.

History-Deterministic Parikh Automata

Enzo Erlich
ENS Rennes, France

Shibashis Guha
Tata Institute of Fundamental Research, Mumbai, India

Ismaël Jecker
University of Warsaw, Poland

Karoliina Lehtinen
CNRS, Aix-Marseille University, LIS, France

Martin Zimmermann
Aalborg University, Denmark

Abstract

Parikh automata extend finite automata by counters that can be tested for membership in a semilinear set, but only at the end of a run. Thereby, they preserve many of the desirable properties of finite automata. Deterministic Parikh automata are strictly weaker than nondeterministic ones, but enjoy better closure and algorithmic properties.

This state of affairs motivates the study of intermediate forms of nondeterminism. Here, we investigate history-deterministic Parikh automata, i.e., automata whose nondeterminism can be resolved on the fly. This restricted form of nondeterminism is well-suited for applications which classically call for determinism, e.g., solving games and composition.

We show that history-deterministic Parikh automata are strictly more expressive than deterministic ones, incomparable to unambiguous ones, and enjoy almost all of the closure properties of deterministic automata.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Parikh automata, History-determinism, Reversal-bounded Counter Machines

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.31

Related Version *Full Version*: https://arxiv.org/abs/2209.07745 [13]

Funding *Shibashis Guha*: Supported by Indian Science and Engineering Research Board (SERB) grant SRG/2021/000466.
Ismaël Jecker: Supported by the ERC grant 950398 (INFSYS).
Martin Zimmermann: Supported by DIREC – Digital Research Centre Denmark.

1 Introduction

Some of the most profound (and challenging) questions of theoretical computer science are concerned with the different properties of deterministic and nondeterministic computation, the P vs. NP problem being arguably the most important and surely the most well-known one. However, even in the more modest setting of automata theory, there is a tradeoff between deterministic and nondeterministic automata with far-reaching consequences for, e.g., the automated verification of finite-state systems. In the automata-based approach to model checking, for example, one captures a finite-state system $\mathcal{S}$ and a specification φ by automata $\mathcal{A}_{\mathcal{S}}$ and $\mathcal{A}_\varphi$ and then checks whether $L(\mathcal{A}_{\mathcal{S}}) \subseteq L(\mathcal{A}_\varphi)$ holds, i.e., whether every execution of $\mathcal{S}$ satisfies the specification φ. To do so, one tests $L(\mathcal{A}_{\mathcal{S}}) \cap \overline{L(\mathcal{A}_\varphi)}$ for emptiness. Hence, one is interested in expressive automata models that have good closure and algorithmic properties. Nondeterminism yields conciseness (think DFA's vs. NFA's) or

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 31; pp. 31:1–31:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

even more expressiveness (think pushdown automata) while deterministic automata often have better algorithmic properties and better closure properties (again, think, e.g., pushdown automata).

Limited forms of nondeterminism constitute an appealing middle ground as they often combine the best of both worlds, e.g., increased expressiveness in comparison to deterministic automata and better algorithmic and closure properties than nondeterministic ones. A classical, and well-studied, example are unambiguous automata, i.e., nondeterministic automata that have at most one accepting run for every input. For example, unambiguous finite automata can be exponentially smaller than deterministic ones while unambiguous pushdown automata are more expressive than deterministic ones [25].

Another restricted class of nondeterministic automata is that of residual automata [12], automata where every state accepts a residual language of the automaton's language. For every regular language there exists a residual automaton. While there exist residual automata that can be exponentially smaller than DFA, there also exist languages for which NFA can be exponentially smaller than residual automata [12].

More recently, another notion of limited nondeterminism has received considerable attention: history-deterministic automata [9, 24][1] are nondeterministic automata whose nondeterminism can be resolved based on the run constructed thus far, but independently of the remainder of the input. This property makes history-deterministic automata suitable for the composition with games, trees, and other automata, applications which classically require deterministic automata. History-determinism has been studied in the context of regular [1, 24, 28], pushdown [22, 29], quantitative [3, 9], and timed automata [23]. For automata that can be determinized, history-determinism offers the potential for succinctness (e.g., co-Büchi automata [28]) while for automata that cannot be determinized, it even offers the potential for increased expressiveness (e.g., pushdown automata [22, 29]). In the quantitative setting, the exact power of history-determinism depends largely on the type of quantitative automata under consideration. So far, it has been studied for quantitative automata in which runs accumulate weights into a value using a value function such as `Sum, LimInf, Average`, and that assign to a word the supremum among the values of its runs. For these automata, history-determinism turns out to have interesting applications for quantitative synthesis [2]. Here, we continue this line of work by investigating history-deterministic Parikh automata, a mildly quantitative form of automata.

Parikh automata, introduced by Klaedtke and Rueß [27], consist of finite automata, augmented with counters that can only be incremented. A Parikh automaton only accepts a word if the final counter-configuration is within a semilinear set specified in the automaton. As the counters do not interfere with the control flow of the automaton, that is, counter values do not affect whether transitions are enabled, they allow for mildly quantitative computations without the full power of vector addition systems or other more powerful models.

For example the language of words over the alphabet $\{0, 1\}$ having a prefix with strictly more 1's than 0's is accepted by a Parikh automaton that starts by counting the number of 0's and 1's and after some prefix nondeterministically stops counting during the processing of the input. It accepts if the counter counting the 1's is, at the end of the run, indeed larger than the counter counting the 0's. Note that the nondeterministic choice can be made based on the word processed so far, i.e., as soon as a prefix with more 1's than 0's is encountered, the counting is stopped. Hence, the automaton described above is in fact history-deterministic.

[1] There is a closely related notion, good-for-gameness, which is often, but not always equivalent [2] (despite frequently being used interchangeably in the past).

Klaedtke and Rueß [27] showed Parikh automata to be expressively equivalent to a quantitative version of existential weak MSO that allows for reasoning about set cardinalities. Their expressiveness also coincides with that of reversal-bounded counter machines [27], in which counters can go from decrementing to incrementing only a bounded number of times, but in which counters affect control flow [26]. The weakly unambiguous restriction of Parikh automata, that is, those that have at most one accepting run, on the other hand, coincide with unambiguous reversal-bounded counter machines [4]. Parikh automata are also expressively equivalent to weighted finite automata over the groups $(\mathbb{Z}^k, +, 0)$ [11, 31] for $k \geqslant 1$. This shows that Parikh automata accept a natural class of quantitative specifications.

Despite their expressiveness, Parikh automata retain some decidability: nonemptiness, in particular, is NP-complete [14]. For weakly unambiguous Parikh automata, inclusion [7] and regular separability [8] are decidable as well. Figueira and Libkin [14] also argued that this model is well-suited for querying graph databases, while mitigating some of the complexity issues related with more expressive query languages. Further, they have been used in the model checking of transducer properties [16].

As Parikh automata have been established as a robust and useful model, many variants thereof exist: pushdown (visibly [10] and otherwise [32]), two-way with [10] and without stack [15], unambiguous [6], and weakly unambiguous [4] Parikh automata, to name a few.

Our contribution. We introduce history-deterministic Parikh automata (HDPA) and study their expressiveness, their closure properties, and their algorithmic properties.

Our main result shows that history-deterministic Parikh automata are more expressive than deterministic ones (DPA), but less expressive than nondeterministic ones (PA). Furthermore, we show that they are of incomparable expressiveness to both classes of unambiguous Parikh automata found in the literature, but equivalent to history-deterministic reversal-bounded counter machines, another class of history-deterministic automata that is studied here for the first time. These results show that history-deterministic Parikh automata indeed constitute a novel class of languages capturing quantitative features.

Secondly, we show that history-deterministic Parikh automata satisfy almost the same closure properties as deterministic ones, the only difference being non-closure under complementation. This result has to be contrasted with unambiguous Parikh automata being closed under complement [6]. Thus, history-determinism is a too strong form of nondeterminism to preserve closure under complementation, a phenomenon that has already been observed in the case of pushdown automata [22, 29].

Finally, we study the algorithmic properties of history-deterministic Parikh automata. Most importantly, safety model checking for HDPA is decidable, as it is for PA. The problem asks, given a system and a set of *bad* prefixes specified by an automaton, whether the system has an execution that has a bad prefix. This allows, for example, to check properties of an arbiter of some shared resource like “the accumulated waiting time between requests and responses of client 1 is always at most twice the accumulated waiting time for client 2 and vice versa”. Note that this property is not ω-regular.

Non-emptiness and finiteness are also both decidable for HDPA (as they are for nondeterministic automata), but universality, inclusion, equivalence, and regularity are not. This is in stark contrast to unambiguous Parikh automata (and therefore also deterministic ones), for which *all* of these problems are decidable. Finally, we show that it is undecidable whether a Parikh automaton is history-deterministic and whether it is equivalent to a history-deterministic one.

Note that we consider only automata over finite words here, but many of our results can straightforwardly be transferred to Parikh automata over infinite words, introduced independently by Guha et al. [21] and Grobler et al. [18, 19].

All proofs omitted due to space restrictions can be found in [13].

2 Definitions

An alphabet is a finite nonempty set Σ of letters. As usual, ε denotes the empty word, Σ^* denotes the set of finite words over Σ, Σ^+ denotes the set of finite nonempty words over Σ, and Σ^ω denotes the set of infinite words over Σ. The length of a finite word w is denoted by $|w|$ and, for notational convenience, we define $|w| = \infty$ for all infinite words w. Finally, $|w|_a$ denotes the number of occurrences of the letter a in a finite word w.

Semilinear Sets. We denote the set of nonnegative integers by $\mathbb{N}$. Given vectors $\vec{v} = (v_0, \ldots, v_{d-1}) \in \mathbb{N}^d$ and $\vec{v}\,' = (v'_0, \ldots, v'_{d'-1}) \in \mathbb{N}^{d'}$, we define their concatenation $\vec{v} \cdot \vec{v}\,' = (v_0, \ldots, v_{d-1}, v'_0, \ldots, v'_{d'-1}) \in \mathbb{N}^{d+d'}$. We lift the concatenation of vectors to sets $D \subseteq \mathbb{N}^d$ and $D' \subseteq \mathbb{N}^{d'}$ via $D \cdot D' = \{\vec{v} \cdot \vec{v}\,' \mid \vec{v} \in D \text{ and } \vec{v}\,' \in D'\}$.

Let $d \geqslant 1$. A set $C \subseteq \mathbb{N}^d$ is linear if there are vectors $\vec{v}_0, \ldots, \vec{v}_k \in \mathbb{N}^d$ such that

$$C = \left\{ \vec{v}_0 + \sum\nolimits_{i=1}^{k} c_i \vec{v}_i \;\middle|\; c_i \in \mathbb{N} \text{ for } i = 1, \ldots, k \right\}.$$

Furthermore, a subset of $\mathbb{N}^d$ is semilinear if it is a finite union of linear sets.

▶ **Example 1.** The sets $\{(n,n) \mid n \in \mathbb{N}\} = \{(0,0) + c(1,1) \mid c \in \mathbb{N}\}$ and $\{(n,2n) \mid n \in \mathbb{N}\} = \{(0,0) + c(1,2) \mid c \in \mathbb{N}\}$ are linear, so their union is semilinear. Further, the set $\{(n,n') \mid n < n'\} = \{(0,1) + c_1(1,1) + c_2(0,1) \mid c_1, c_2 \in \mathbb{N}\}$ is linear and thus also semilinear.

▶ **Proposition 2** ([17]). *If $C, C' \subseteq \mathbb{N}^d$ are semilinear, then so are $C \cup C'$, $C \cap C'$, $\mathbb{N}^d \setminus C$, as well as $\mathbb{N}^{d'} \cdot C$ and $C \cdot \mathbb{N}^{d'}$ for every $d' \geqslant 1$.*

Finite Automata. A (nondeterministic) finite automaton (NFA) $\mathcal{A} = (Q, \Sigma, q_I, \Delta, F)$ over Σ consists of the finite set Q of states containing the initial state q_I, the alphabet Σ, the transition relation $\Delta \subseteq Q \times \Sigma \times Q$, and the set $F \subseteq Q$ of accepting states. The NFA is deterministic (i.e., a DFA) if for every state $q \in Q$ and every letter $a \in \Sigma$, there is at most one $q' \in Q$ such that (q, a, q') is a transition of $\mathcal{A}$.

A run of $\mathcal{A}$ is a (possibly empty) sequence $(q_0, a_0, q_1)(q_1, a_1, q_2) \cdots (q_{n-1}, a_{n-1}, q_n)$ of transitions with $q_0 = q_I$. It processes the word $a_0 a_1 \cdots a_{n-1} \in \Sigma^*$. The run is accepting if it is either empty and the initial state is accepting or if it is nonempty and q_n is accepting. The language $L(\mathcal{A})$ of $\mathcal{A}$ contains all finite words $w \in \Sigma^*$ such that $\mathcal{A}$ has an accepting run processing w.

Parikh Automata. Let Σ be an alphabet, $d \geqslant 1$, and D a finite subset of $\mathbb{N}^d$. Furthermore, let $w = (a_0, \vec{v}_0) \cdots (a_{n-1}, \vec{v}_{n-1})$ be a word over $\Sigma \times D$. The Σ-projection of w is $p_\Sigma(w) = a_0 \cdots a_{n-1} \in \Sigma^*$ and its *extended Parikh image* is $\Phi_e(w) = \sum_{j=0}^{n-1} \vec{v}_j \in \mathbb{N}^d$ with the convention $\Phi_e(\varepsilon) = \vec{0}$, where $\vec{0}$ is the d-dimensional zero vector.

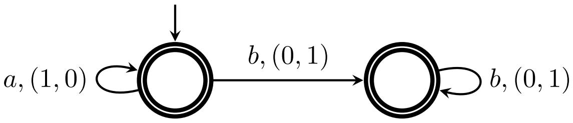


Figure 1 The automaton for Example 3.

A Parikh automaton (PA) is a pair $(\mathcal{A}, C)$ such that $\mathcal{A}$ is an NFA over $\Sigma \times D$ for some input alphabet Σ and some finite $D \subseteq \mathbb{N}^d$ for some $d \geqslant 1$, and $C \subseteq \mathbb{N}^d$ is semilinear. The language of $(\mathcal{A}, C)$ consists of the Σ-projections of words $w \in L(\mathcal{A})$ whose extended Parikh image is in C, i.e.,

$$L(\mathcal{A}, C) = \{p_\Sigma(w) \mid w \in L(\mathcal{A}) \text{ with } \Phi_e(w) \in C\}.$$

The Parikh automaton $(\mathcal{A}, C)$ is deterministic if for every state q of $\mathcal{A}$ and every $a \in \Sigma$, there is at most one pair $(q', \vec{v}) \in Q \times D$ such that $(q, (a, \vec{v}), q')$ is a transition of $\mathcal{A}$. Note that this definition does *not* coincide with $\mathcal{A}$ being a DFA: As mentioned above, $\mathcal{A}$ accepts words over $\Sigma \times D$ while $(\mathcal{A}, C)$ accepts words over Σ. Therefore, determinism is defined with respect to Σ only.

Note that the above definition of $L(\mathcal{A}, C)$ coincides with the following alternative definition via accepting runs: A run ρ of $(\mathcal{A}, C)$ is a run

$$\rho = (q_0, (a_0, \vec{v}_0), q_1)(q_1, (a_1, \vec{v}_1), q_2) \cdots (q_{n-1}, (a_{n-1}, \vec{v}_{n-1}), q_n)$$

of $\mathcal{A}$. We say that ρ *processes* the word $a_0 a_1 \cdots a_{n-1} \in \Sigma^*$, i.e., the $\vec{v}_j$ are ignored, and that ρ's extended Parikh image is $\sum_{j=0}^{n-1} \vec{v}_j$. The run is accepting if it is either empty and both the initial state of $\mathcal{A}$ is accepting and the zero vector (the extended Parikh image of the empty run) is in C, or if it is nonempty, q_n is accepting, and ρ's extended Parikh image is in C. Finally, $(\mathcal{A}, C)$ accepts $w \in \Sigma^*$ if it has an accepting run processing w.

▶ **Example 3.** Consider the deterministic PA $(\mathcal{A}, C)$ with $\mathcal{A}$ in Figure 1 and $C = \{(n,n) \mid n \in \mathbb{N}\} \cup \{(n, 2n) \mid n \in \mathbb{N}\}$ (cf. Example 1). It accepts the language $\{a^n b^n \mid n \in \mathbb{N}\} \cup \{a^n b^{2n} \mid n \in \mathbb{N}\}$.

3 History-deterministic Parikh Automata

In this section, we introduce history-deterministic Parikh automata and give examples.

Let $(\mathcal{A}, C)$ be a PA with $\mathcal{A} = (Q, \Sigma \times D, q_I, \Delta, F)$. For a function $r \colon \Sigma^+ \to \Delta$ we define its iteration $r^* \colon \Sigma^* \to \Delta^*$ via $r^*(\varepsilon) = \varepsilon$ and $r^*(a_0 \cdots a_n) = r^*(a_0 \cdots a_{n-1}) \cdot r(a_0 \cdots a_n)$. We say that r is a resolver for $(\mathcal{A}, C)$ if, for every $w \in L(\mathcal{A}, C)$, $r^*(w)$ is an accepting run of $(\mathcal{A}, C)$ processing w. Further, we say that $(\mathcal{A}, C)$ is history-deterministic (i.e., an HDPA) if it has a resolver.

▶ **Example 4.** Fix $\Sigma = \{0, 1\}$ and say that a word $w \in \Sigma^*$ is non-Dyck if $|w|_0 < |w|_1$. We consider the language $N \subseteq \Sigma^+$ of words that have a non-Dyck prefix. It is accepted by the PA $(\mathcal{A}, C)$ where $\mathcal{A}$ is depicted in Figure 2 and $C = \{(n, n') \mid n < n'\}$ (cf. Example 1). Intuitively, in the initial state q_c, the automaton counts the number of 0's and 1's occurring in some prefix, nondeterministically decides to stop counting by moving to q_n (this is the only nondeterminism in $\mathcal{A}$), and accepts if there are more 1's than 0's in the prefix.

The nondeterministic choice can be made only based on the prefix processed so far, i.e., as soon as the first non-Dyck prefix is encountered, the resolver proceeds to state q_n, thereby ending the prefix. Formally, the function

$$wb \mapsto \begin{cases} (q_c, (b, (1-b, b)), q_c) & \text{if } wb \text{ has no non-Dyck prefix,} \\ (q_c, (b, (1-b, b)), q_n) & \text{if } wb \text{ is non-Dyck, but } w \text{ has no non-Dyck prefix,} \\ (q_n, (b, (0, 0)), q_n) & \text{if } w \text{ has a non-Dyck prefix,} \end{cases}$$

is a resolver for $(\mathcal{A}, C)$.

▶ Remark 5. As a resolver resolves nondeterminism and a DPA has no nondeterminism to resolve, every DPA is history-deterministic.

4 Expressiveness

In this section, we study the expressiveness of HDPA by comparing them to related automata models, e.g., deterministic and nondeterministic Parikh automata, unambiguous Parikh automata (capturing another restricted notion of nondeterminism), and reversal-bounded counter machines (which are known to be related to Parikh automata). Overall, we obtain the relations shown in Figure 3, where the additional classes of languages and the separating languages will be introduced throughout this section.

We begin by stating and proving a pumping lemma for HDPA. The basic property used here, just as for the pumping lemmata for PA and DPA [5], is that shuffling around cycles of a run does not change whether it is accepting or not, as acceptance only depends on the last state of the run being accepting and the vectors (and their multiplicity) that appear on the run, but not the order of their appearance.

▶ **Lemma 6.** *Let $(\mathcal{A}, C)$ be an HDPA with $L(\mathcal{A}, C) \subseteq \Sigma^*$. Then, there exist $p, \ell \in \mathbb{N}$ such that every $w \in \Sigma^*$ with $|w| > \ell$ can be written as $w = uvxvz$ such that*

- *$0 < |v| \leqslant p$, $|x| > p$, and $|uvxv| \leqslant \ell$, and*
- *for all $z' \in \Sigma^*$: if $uvxvz' \in L(\mathcal{A}, C)$, then also $uv^2xz' \in L(\mathcal{A}, C)$ and $uxv^2z' \in L(\mathcal{A}, C)$.*

Proof. Fix some resolver r for $(\mathcal{A}, C)$. Note that the definition of a resolver only requires $r^*(w)$ to be a run processing w for those $w \in L(\mathcal{A}, C)$. Here, we assume without loss of generality that $r^*(w)$ is a run processing w for each $w \in \Sigma^*$. This can be achieved by completing $\mathcal{A}$ (by adding a nonaccepting sink state and transitions to the sink where necessary) and redefining r where necessary (which is only the case for inputs that cannot be extended to a word in $L(\mathcal{A}, C)$).

A cycle is a nonempty finite run infix

$$(q_0, a_0, q_1)(q_1, a_1, q_2) \cdots (q_{n-1}, a_{n-1}, q_n)(q_n, a_n, q_0)$$

starting and ending in the same state and such that the q_j are pairwise different. Now, let p be the number of states of $\mathcal{A}$ and let m be the number of cycles of $\mathcal{A}$. Note that every run infix containing at least p transitions contains a cycle.

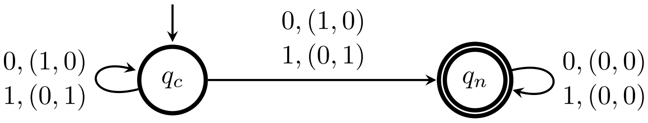


Figure 2 The automaton for Example 4.

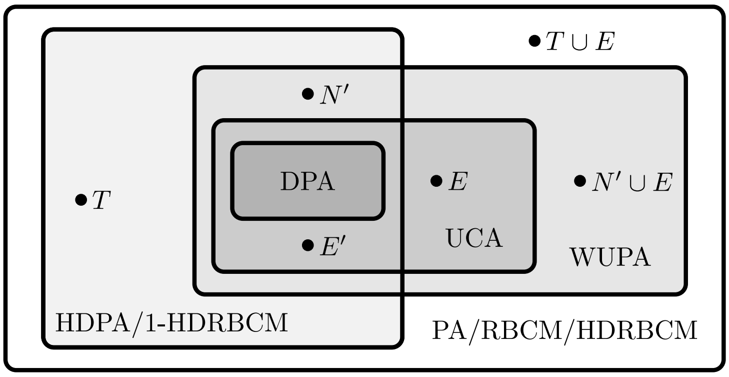


Figure 3 The classes of languages accepted by different models of Parikh automata.

We define $\ell = (p+1)(2m+1)$, consider a word $w \in \Sigma^*$ with $|w| > \ell$, and let $\rho = r^*(w)$ be the run of $\mathcal{A}$ induced by r which processes w. We split ρ into $\rho_0\rho_1 \cdots \rho_{2m}\rho'$ such that each ρ_j contains $p+1$ transitions. Then, each ρ_j contains a cycle and there are j_0, j_1 with $j_1 > j_0 + 1$ such that ρ_{j_0} and ρ_{j_1} contain the same cycle. Now, let

- ρ_v be the cycle in ρ_{j_0} and ρ_{j_1},
- ρ_u be the prefix of ρ before the first occurrence of ρ_v in ρ_{j_0}, and
- ρ_x be the infix of ρ between the first occurrences of ρ_v in ρ_{j_0} and ρ_{j_1}.

Furthermore, let $u, v, x \in \Sigma^*$ be the inputs processed by ρ_u, ρ_v, and ρ_x respectively. Then, we indeed have $0 < |v| \leqslant p$ (as we consider simple cycles), $|x| > p$ (as $j_1 > j_0 + 1$), and $|uvxv| \leqslant \ell$.

Note that $\rho_u\rho_v\rho_x\rho_v$, $\rho_u\rho_v^2\rho_x$, and $\rho_u\rho_x\rho_v^2$ are all runs of $\mathcal{A}$ which process $uvxv$, uv^2x, and uxv^2 respectively. Furthermore, all three runs end in the same state and their extended Parikh images are equal, as we only shuffled pieces around.

Now, consider some z' such that $uvxvz' \in L(\mathcal{A}, C)$. Then $r^*(uvxvz')$ is an accepting run, and of the form $\rho_u\rho_v\rho_x\rho_v\rho_{z'}$ for some $\rho_{z'}$ processing z'. Now, $\rho_u\rho_v^2\rho_x\rho_{z'}$, and $\rho_u\rho_x\rho_v^2\rho_{z'}$ are accepting runs of $(\mathcal{A}, C)$ (although not necessarily induced by r) processing uv^2xz' and uxv^2z', respectively. Thus, $uv^2xz' \in L(\mathcal{A}, C)$ and $uxv^2z' \in L(\mathcal{A}, C)$. ◀

It is instructive to compare our pumping lemma for HDPA to those for PA and DPA [5]:

- The pumping lemma for PA states that every *long* word accepted by a PA can be decomposed into $uvxvz$ as above such that both uv^2xz and uxv^2z are accepted as well. This statement is weaker than ours, as it only applies to the two words obtained by moving a v while our pumping lemma applies to any suffix z'. This is possible, as the runs of an HDPA on words of the form $uvxvz'$ (for fixed $uvxv$), induced by a resolver, all coincide on their prefixes processing $uvxv$. This is not necessarily the case in PA.
- The pumping lemma for DPA states that every *long* word (not necessarily accepted by the automaton) can be decomposed into $uvxvz$ as above such that $uvxv$, uv^2x, and uxv^2 are all equivalent with respect to the Myhill-Nerode equivalence. This statement is stronger than ours, as Myhill-Nerode equivalence is concerned both with the language of the automaton and its complement. But similarly to our pumping lemma, the one for DPA applies to all possible suffixes z'.

Now, we apply the pumping lemma to compare the expressiveness of HDPA, DPA, and PA.

▶ **Theorem 7.** *HDPA are more expressive than DPA, but less expressive than PA.*

Proof. First, we consider the separation between DPA and HDPA. The language N from Example 4, which is accepted by an HDPA, is known to be not accepted by any DPA: DPA are closed under complementation [27] while the complement of N is not even accepted by any PA [6].

To show that PA are more expressive than HDPA, consider the language $E = \{a, b\}^* \cdot \{a^n b^n \mid n > 0\}$, which can easily be seen to be accepted by a PA. We show that E is not accepted by any HDPA[2] via an application of the pumping lemma.

To this end, assume there is some HDPA $(\mathcal{A}, C)$ accepting E, and let p, ℓ as in the pumping lemma. We pick $w = (a^{p+1}b^{p+1})^\ell$, which we decompose as $uvxvz$ with the properties guaranteed by the pumping lemma. In particular, we have $|v| \leqslant p$, therefore $v \in a^*b^* + b^*a^*$. We consider two cases depending on the last letter of v. In each one, we show the existence of a word z' such that the word $uvxvz'$ is in the language E, yet either uv^2xz' or uxv^2z' is not. This yields the desired contradiction to the pumping lemma.

1. First, assume that the last letter of v is an a. Since $|x| > p$ and x appears between two copies of v in $(a^{p+1}b^{p+1})^\ell$, the infix xv contains at least one full b-block: we have $xv = x'b^{p+1}a^k$ with $x' \in \{a, b\}^*$ and $0 < k \leqslant p + 1$. We set $z' = a^{p+1-k}b^{p+1}$. Hence, $uvxvz' = uvx'b^{p+1}a^{p+1}b^{p+1} \in E$. We show that $uv^2xz' \notin E$ by differentiating two cases:
   - **a.** If $v = a^i$ for some i, which must satisfy $0 < i \leqslant k$, then uv^2xz' is not in E as it ends with $b^{p+1}a^{p+1-i}b^{p+1}$.
   - **b.** Otherwise, we must have $v = b^i a^k$ with $0 < i < p$. Then, uv^2xz' is not in E as it ends with $b^{p+1-i}a^{p+1-k}b^{p+1}$.
2. Otherwise, the last letter of v is a b. Since $|x| > p$ and x appears between two copies of v in $(a^{p+1}b^{p+1})^\ell$, the infix xv contains at least one full a-block: we have $xv = x'a^{p+1}b^k$ with $x' \in \{a, b\}^*$ and $0 < k \leqslant p + 1$. This time we set $z' = b^{p+1-k}$. Thus, $uvxvz' = uvx'a^{p+1}b^{p+1} \in E$, and we differentiate two cases to show that $uxv^2z' \notin E$:
   - **a.** If $v = b^i$ for some i, which must satisfy $0 < i \leqslant p$, then uxv^2z' ends with b^{p+i+1}. However, each of its a-blocks has length $p + 1$, as moving $v = b^i$ with $i \leqslant p$ does not merge any a-blocks. Hence, uxv^2z' is not in E.
   - **b.** Otherwise, we must have $v = a^i b^k$ with $0 < i < p$. Then, uxv^2z' is not in E as it ends with $v^2z' = a^i b^k a^i b^{p+1}$. ◀

4.1 History-determinism vs. Unambiguity

After having placed history-deterministic Parikh automata strictly between deterministic and nondeterministic ones, we now compare them to unambiguous Parikh automata, another class of automata whose expressiveness lies strictly between that of DPA and PA. In the literature, there are two (nonequivalent) forms of unambiguous Parikh automata. We consider both of them here.

Cadilhac et al. studied unambiguity in Parikh automata in the guise of unambiguous constrained automata (UCA) [6]. Constrained automata are a related model and effectively equivalent to PA. Intuitively, an UCA $(\mathcal{A}, C)$ over an alphabet Σ consists of an unambiguous ε-NFA $\mathcal{A}$ over Σ, say with d transitions, and a semilinear set $C \subseteq \mathbb{N}^d$, i.e., C has one dimension for each transition in $\mathcal{A}$. It accepts a word $w \in \Sigma^*$ if $\mathcal{A}$ has an accepting run processing w (due to unambiguity this run must be unique) such that the Parikh image of the run (recording the number of times each transition occurs in the run) is in C.

[2] Note that the related language $\{a, b\}^* \cdot \{a^n\#a^n \mid n \in \mathbb{N}\}$ is not accepted by any DPA [5].

On the other hand, Bostan et al. introduced so-called weakly-unambiguous Parikh automata (WUPA) [4]. Intuitively, a WUPA $(\mathcal{A}, C)$ over Σ is a classical PA as introduced here where every input over Σ has at most one accepting run (in the sense of the definition in Section 2). Bostan et al. discuss the different definitions of unambiguity and in particular show that every UCA is a WUPA, but that WUPA are strictly more expressive [4]. Here, we compare the expressiveness of HDPA to that of UCA and WUPA.

The language E from the proof of Theorem 7 is accepted by an UCA [6] and a WUPA [4], but not by any HDPA (see Theorem 7). On the other hand, the language

$$T = \{c^{n_0}dc^{n_1}d \cdots c^{n_k}d \mid k \geq 1, n_0 = 1, \text{ and } n_{j+1} \neq 2n_j \text{ for some } 0 \leq j < k\}$$

is not accepted by any WUPA, and hence also not by any UCA [4], but there is an HDPA accepting it. Hence, these two languages show that these three classes of automata have incomparable expressiveness.

▶ **Theorem 8.** *The expressiveness of HDPA is neither comparable with that of UCA nor with that of WUPA.*

Finally, we show that all intersections between the different classes introduced above are nonempty.

▶ **Theorem 9.**

1. *There is a language that is accepted by an HDPA and by an UCA, but not by any DPA.*
2. *There is a language that is accepted by an HDPA and by a WUPA, but not by any UCA.*
3. *There is a language that is accepted by a PA, but not by any HDPA nor by any WUPA.*
4. *There is a language that is accepted by a WUPA, but not by any HDPA nor by any UCA.*

Here, we give proof sketches, full proofs can be found in [13].

Recall that the language $E = \{a, b\}^* \cdot \{a^n b^n \mid n > 0\}$ from Theorem 7 is accepted by an UCA, but not by any HDPA. However, a slight modification allows it to be accepted by both types of automata, but not by a deterministic automaton: We show that

$$E' = \{c^m \{a, b\}^{m-1} b a^n b^n \mid m, n > 0\}$$

has the desired property. Intuitively, the prefix c^m allows us to resolve the nondeterminism on-the-fly, but nondeterminism is still required.

The second separation relies on a similar trick: The language

$$N' = \{c^n w \mid w = a_0 \cdots a_k \in \{0, 1\}^*, |w| \geq n > 0, \text{ and } a_0 \cdots a_{n-1} \text{ is non-Dyck}\},$$

which is a variation of the language N of words that have a non-Dyck prefix, was shown to be accepted by a WUPA, but not by any UCA [4]. It is straightforward to show that it is also accepted by an HDPA.

The last two results follow easily from closure properties: The union $T \cup E$ is neither accepted by any HDPA nor by any WUPA, as both models are closed under intersection[3], i.e., $(T \cup E) \cap \{a, b\}^* = E$ and $(T \cup E) \cap \{c, d\}^* = T$ yield the desired separation. A similar argument works for $N' \cup E$, which is accepted by some WUPA (as WUPA are closed under disjoint unions) but not by any HDPA nor by any UCA, as both classes are closed under intersection.

[3] For WUPA, this was shown by Bostan et al. [4], for HDPA this is shown in the next section.

4.2 History-deterministic Reversal-bounded Counter Machines

There is one more automaton model that is closely related to Parikh automata, i.e., reversal-bounded counter machines, originally introduced by Ibarra [26]. These are, in their most general form, two-way automata with multiple counters that can be incremented, decremented, and tested for zero, but there is a constant bound on the number of reversals of the reading head *and* on the number of switches between increments and decrements (on each counter). It is known that Parikh automata and nondeterministic reversal-bounded counter machines are equivalent [27], while deterministic reversal-bounded counter machines are strictly more expressive than deterministic Parikh automata [5]. Here, we compare history-deterministic reversal-bounded counter machines and history-deterministic Parikh automata (and, for technical reasons, also history-deterministic Parikh automata with ε-transitions).

We begin by introducing counter machines and then their reversal-bounded variant.

A (two-way) counter machine is a tuple $\mathcal{M} = (k, Q, \Sigma, \triangleright, \triangleleft, q_I, \Delta, F)$ where $k \in \mathbb{N}$ is the number of counters, Q is the finite set of states, Σ is the alphabet, $\triangleright, \triangleleft \notin \Sigma$ are the left and right endmarkers respectively, $q_I \in Q$ is the initial state,

$$\Delta \subseteq (Q \times \Sigma_{\triangleright\triangleleft} \times \{0,1\}^k) \times (Q \times \{-1,0,1\} \times \{-1,0,1\}^k)$$

is the transition relation, and $F \subseteq Q$ is the set of accepting states. Here, we use the shorthand $\Sigma_{\triangleright\triangleleft} = \Sigma \cup \{\triangleright, \triangleleft\}$. Intuitively, a transition $((q, a, \vec{g}), (q', m, \vec{v}))$ is enabled if the current state is q, the current letter on the tape is a, and for each $0 \leq j \leq k-1$, the j-th entry in the guard $\vec{g}$ is nonzero if and only if the current value of counter j is nonzero. Taking this transition updates the state to q', moves the head in direction m, and adds the j-th entry of $\vec{v}$ to counter j.

We require that all transitions $((q, a, \vec{g}), (q', m, \vec{v})) \in \Delta$ satisfy the following properties:

- If $a = \triangleright$, then $m \geq 0$: the head never leaves the tape to the left.
- If $a = \triangleleft$, then $m \leq 0$: the head never leaves the tape to the right.
- $\vec{g}$ and $\vec{v}$ are *compatible*, i.e. if the j-th entry of $\vec{g}$ is zero, then the j-th entry of $\vec{v}$ is nonnegative: a zero counter is not decremented.

For the sake of brevity, we refer the formal definition of the semantics to the full version [13].

In the following, we just need the definition of configurations: A configuration of $\mathcal{M}$ on an input $w \in \Sigma^*$ is of the form $(q, \triangleright w \triangleleft, h, \vec{c})$ where $q \in Q$ is the current state, $\triangleright w \triangleleft$ is the content of the tape (which does not change during a run), $0 \leq h \leq |w| + 1$ is the current position of the reading head, and $\vec{c} \in \mathbb{N}^k$ is the vector of current counter values.

We say that a two-way counter machine is reversal-bounded, if there is a $b \in \mathbb{N}$ such that on each run, the reading head reverses its direction at most b times *and* each counter switches between incrementing and decrementing at most b times. We write RBCM for reversal-bounded counter machines and 1-RBCM for RBCM that do not make a reversal of the reading head (i.e., they are one-way). Their deterministic variants are denoted by DRBCM and 1-DRBCM, respectively.

Ibarra [26] has shown that every RBCM can be effectively turned into an equivalent 1-RBCM and that every RBCM can be effectively turned into an equivalent one where the number of reversals of each counter is bounded by 1. The latter construction preserves determinism and one-wayness. Hence, in the following, we assume that during each run of an RBCM, each counter reverses at most once.

In terms of expressiveness, Klaedtke and Rueß [27] showed that RBCM are equivalent to Parikh automata while Cadilhac et al. [5] showed that 1-DRBCM are strictly more expressive than DPA.

In the following, we determine the relation between history-deterministic RBCM and HDPA. To this end, we first have to define the notion of history-determinism for RBCM, which is slightly technical due to the two-wayness of these machines.

Let $\mathcal{M} = (k, Q, \Sigma, \triangleright, \triangleleft, q_I, \Delta, F)$ be an RBCM. Given a sequence $\tau_0 \cdots \tau_j$ of transitions inducing a run ρ, let $\mathrm{pos}(\tau_0 \cdots \tau_j)$ be the position of the reading head at the end of ρ, so in particular $\mathrm{pos}(\varepsilon) = 0$. Hence, $(\triangleright w \triangleleft)_{\mathrm{pos}(\tau_0\tau_1\cdots\tau_j)}$ is the letter the reading head is currently pointing to.

A resolver for $\mathcal{M}$ is a function $r \colon \Delta^* \times \Sigma_{\bowtie} \to \Delta$ such that if w is accepted by $\mathcal{M}$, there is a sequence of transitions $\tau_0\tau_1 \cdots \tau_{n-1}$ such that

- $\tau_{j+1} = r(\tau_0\tau_1 \cdots \tau_j, (\triangleright w \triangleleft)_{\mathrm{pos}(\tau_0\tau_1\cdots\tau_j)})$ for all $0 \le j < n - 1$, and
- the run of $\mathcal{M}$ on w induced by the sequence of transitions $\tau_0\tau_1 \cdots \tau_{n-1}$ is accepting.

An RBCM $\mathcal{M}$ is history-deterministic (an HDRBCM) if there exists a resolver for $\mathcal{M}$. One-way HDRBCM are denoted by 1-HDRBCM.

Now, we are able to state the main theorem of this subsection: History-deterministic two-way RBCM are as expressive as RBCM and PA while history-deterministic one-way RBCM are as expressive as history-deterministic PA.

▶ **Theorem 10.**

1. *HDRBCM are as expressive as RBCM, and therefore as expressive as PA.*
2. *1-HDRBCM are as expressive as HDPA.*

The proof of the first equivalence is very general and not restricted to RBCM: A two-way automaton over finite inputs can first read the whole input and then resolve nondeterministic choices based on the whole word. Spelt out more concisely: two-wayness makes history-determinism as powerful as general nondeterminism.

For the other equivalence, both directions are nontrivial: We show how to simulate a PA using an RBCM while preserving history-determinism, and how to simulate a 1-RBCM by a PA, again while preserving history-determinism. Due to the existence of transitions that do not move the reading head in a 1-RBCM, this simulation takes a detour via PA with ε-transitions.

Finally, let us remark that HDPA (or equivalently 1-HDRBCM) and deterministic RBCM have incomparable expressiveness. Indeed, the language E, which is not accepted by any HDPA (see Theorem 7), can easily be accepted by a deterministic RBCM while the language N (see Example 4) is accepted by an HDPA, but not by any deterministic RBCM [6]. The reason is that these machines are closed under complement, but the complement of N is not accepted by any PA as shown in [6], and therefore also not by any RBCM.

5 Closure Properties

In this subsection, we study the closure properties of history-deterministic Parikh automata, i.e., we consider Boolean operations, concatenation and Kleene star, (inverse) homomorphic image, and commutative closure. Let us begin by recalling the last three notions.

Fix some ordered alphabet $\Sigma = (a_0 < a_1 < \cdots < a_{d-1})$. The Parikh image of a word $w \in \Sigma^*$ is the vector $\Phi(w) = (|w|_{a_0}, |w|_{a_1}, \ldots, |w|_{a_{d-1}})$ and the Parikh image of a language $L \subseteq \Sigma^*$ is $\Phi(L) = \{\Phi(w) \mid w \in L\}$. The commutative closure of L is $\{w \in \Sigma^* \mid \Phi(w) \in \Phi(L)\}$.

Now, fix some alphabets Σ and Γ and a homomorphism $h \colon \Sigma^* \to \Gamma^*$. The homomorphic image of a language $L \subseteq \Sigma^*$ is $h(L) = \{h(w) \mid w \in L\} \subseteq \Gamma^*$. Similarly, the inverse homomorphic image of a language $L \subseteq \Gamma^*$ is $h^{-1}(L) = \{w \in \Sigma^* \mid h(w) \in L\}$.

Table 1 Closure properties of history-deterministic Parikh automata (in grey) and comparison to other types of Parikh automata (results for other types are from [5, 6, 27]).

	$\cup$	$\cap$	$^-$	$\cdot$	*	h	h^{-1}	c
DPA	Y	Y	Y	N	N	N	Y	Y
HDPA	Y	Y	N	N	N	N	Y	Y
UCA	Y	Y	Y	N	N	N	?	Y
PA	Y	Y	N	Y	N	Y	Y	Y

▶ **Theorem 11.** *HDPA are closed under union, intersection, inverse homomorphic images, and commutative closure, but not under complement, concatenation, Kleene star, and homomorphic image.*

Proof Sketch. Closure under union and intersection is shown by a product construction, while closure under inverse homomorphic images is shown by lifting the construction for finite automata (just as for DPA and PA). Finally closure under commutative closure follows from previous work: Cadilhac et al. proved that the commutative closure of any PA (and therefore that of any HDPA) is accepted by some DPA, and therefore also by some HDPA.

The negative results follow from a combination of expressiveness results proven in Section 4 and nonexpressiveness results in the literature [5, 6]:

- Complement: In the first part of the proof of Theorem 7, we show that the language N is accepted by an HDPA, but its complement is known to not be accepted by any PA [6].
- Concatenation: The language E is the concatenation of the languages $\{a, b\}^*$ and $\{a^n b^n \mid n \in \mathbb{N}\}$, which are both accepted by a DPA, but itself is not accepted by any HDPA (see the proof of Theorem 7).
- Kleene star: There is a DPA (and therefore also an HDPA) such that the Kleene star of its language is not accepted by any PA [5], and therefore also by no HDPA.
- Homomorphic image: There is a DPA (and therefore also an HDPA) and a homomorphism such that the homomorphic image of the DPA's language is not accepted by any PA [5], and therefore also by no HDPA. ◀

Table 1 compares the closure properties of HDPA with those of DPA, UCA, and PA. We do not compare to RBCM, as only deterministic ones differ from Parikh automata and results on these are incomplete: However, Ibarra proved closure under union, intersection, and complement [26].

6 Decision Problems

Next, we study various decision problems for history-deterministic PA. First, let us mention that nonemptiness and finiteness are decidable for HDPA, as these problems are decidable for PA [27, 5]. In the following, we consider universality, inclusion, equivalence, regularity, and model checking. We start with the universality problem.

▶ **Theorem 12.** *The following problem is undecidable: Given an HDPA $(\mathcal{A}, C)$ over Σ, is $L(\mathcal{A}, C) = \Sigma^*$?*

Proof Sketch. The result is shown by turning (deterministic) Minsky machines into HDPA such that the machine does not terminate if and only if the automaton is universal. As nontermination of Minsky machines is undecidable [30], the same is true for HDPA universality.

Intuitively, the automaton processes sequences of instructions of the Minsky machine and checks whether they are prefixes of the unique run of the machine or not, employing the counters of the automaton to simulate the counter of the Minsky machine. Finally, history-determinism can be employed to find a position in the sequence of instructions where it differs from the unique run of the machine. ◀

The next results follow more or less immediately from the undecidability of universality.

▶ **Theorem 13.** *The following problems are undecidable:*

1. *Given two HDPA $(\mathcal{A}_0, C_0)$ and $(\mathcal{A}_1, C_1)$, is $L(\mathcal{A}_0, C_0) \subseteq L(\mathcal{A}_1, C_1)$?*
2. *Given two HDPA $(\mathcal{A}_0, C_0)$ and $(\mathcal{A}_1, C_1)$, is $L(\mathcal{A}_0, C_0) = L(\mathcal{A}_1, C_1)$?*
3. *Given an HDPA $(\mathcal{A}, C)$, is $L(\mathcal{A}, C)$ regular?*
4. *Given an HDPA $(\mathcal{A}, C)$, is $L(\mathcal{A}, C)$ context-free?*

Proof Sketch. The first two items follow directly from the undecidability of universality (cf. Theorem 12), so let us consider the latter two. They are both shown by a variation of the construction proving Theorem 12: Given a Minsky machine we construct an HDPA such that the machine terminates if and only if the automaton accepts a regular language (respectively, a context-free language). ◀

Table 2 compares the decidability of standard problems for HDPA with those of DPA, UCA, and PA.

Finally, we consider the problems of deciding whether a Parikh automaton is history-deterministic and whether it is equivalent to some HDPA. Both of our proofs follow arguments developed for similar results for history-deterministic pushdown automata [29].

▶ **Theorem 14.** *The following problems are undecidable:*

1. *Given a PA $(\mathcal{A}, C)$, is it history-deterministic?*
2. *Given a PA $(\mathcal{A}, C)$, is it equivalent to some HDPA?*

Finally, let us introduce the model-checking problem (for safety properties): A transition system $\mathcal{T} = (V, v_I, E, \lambda)$ consists of a finite set V of vertices containing the initial state $v_I \in V$, a transition relation $E \subseteq V \times V$, and a labeling function $\lambda \colon V \to \Sigma$ for some alphabet Σ. A (finite and initial) path in $\mathcal{T}$ is a sequence $v_0 v_1 \cdots v_n \in V^+$ such that $v_0 = v_I$ and $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$. Infinite (initial) paths are defined analogously. The trace of a path $v_0 v_1 \cdots v_n$ is $\lambda(v_0)\lambda(v_1)\cdots\lambda(v_n) \in \Sigma^+$. We denote the set of traces of paths of $\mathcal{T}$ by $\mathsf{tr}(\mathcal{T})$.

The model-checking problem for HDPA asks, given an HDPA $\mathcal{A}$ and a transition system $\mathcal{T}$, whether $\mathsf{tr}(\mathcal{T}) \cap L(\mathcal{A}) = \emptyset$? Note that the automaton specifies the set of *bad prefixes*, i.e., $\mathcal{T}$ satisfies the specification encoded by $\mathcal{A}$ if no trace of $\mathcal{T}$ is in $L(\mathcal{A})$.

Table 2 Decision problems for history-deterministic Parikh automata (in grey) and comparison to other types of Parikh automata (results are from [5, 6, 27]).

	$\neq \emptyset$?	finite?	$= \Sigma^*$?	$\subseteq$?	$=$?	regular?	MC
DPA	Y	Y	Y	Y	Y	Y	Y
HDPA	Y	Y	N	N	N	N	Y
UCA	Y	Y	Y	Y	Y	Y	Y
PA	Y	Y	N	N	N	N	Y

As the model-checking problem for PA is decidable, so is the model-checking problem for HDPA, which follows from the fact that a transition system $\mathcal{T}$ can be turned into an NFA and hence into a PA $\mathcal{A}_{\mathcal{T}}$ with $L(\mathcal{A}_{\mathcal{T}}) = \mathsf{tr}(\mathcal{T})$. Then, closure under intersection and decidability of nonemptiness yields the desired result.

▶ **Theorem 15.** *The model-checking problem for HDPA is decidable.*

Let us conclude by mentioning that the dual problem, i.e., given a transition system $\mathcal{T}$ and an HDPA $\mathcal{A}$, does every infinite path of $\mathcal{T}$ have a prefix whose trace is in $L(\mathcal{A})$, is undecidable. This follows from recent results on Parikh automata over infinite words [21], i.e., that model checking for Parikh automata with reachability conditions is undecidable. Such automata are syntactically equal to Parikh automata over finite words and an (infinite) run is accepting if it has a prefix ending in an accepting state whose extended Parikh image is in the semilinear set of the automaton.

7 Conclusion

In this work, we have introduced and studied history-deterministic Parikh automata. We have shown that their expressiveness is strictly between that of deterministic and nondeterministic PA, incomparable to that of unambiguous PA, but equivalent to history-deterministic 1-RBCM. Furthermore, we showed that they have almost the same closure properties as DPA (complementation being the notable difference), and enjoy some of the desirable algorithmic properties of DPA.

An interesting direction for further research concerns the complexity of resolving nondeterminism in history-deterministic Parikh automata. It is straightforward to show that every HDPA has a positional resolver (i.e., one whose decision is only based on the last state of the run constructed thus far and on the extended Parikh image induced by this run) and that HDPA that have finite-state resolvers (say, implemented by a Mealy machine) can be determinized by taking the product of the HDPA and the Mealy machine. In fact, both proofs are simple adaptions of the corresponding ones for history-deterministic pushdown automata [22, 29]. A more interesting question is whether there is a notion of Parikh transducer such that every HDPA has a resolver implemented by such a transducer. Note that the analogous result for history-deterministic pushdown automata fails: not every history-deterministic pushdown automaton has a pushdown resolver [22, 29].

Good-for-gameness is another notion of restricted nondeterminism that is very tightly related to history-determinism. In fact, both terms were used interchangeably until very recently, when it was shown that they do not always coincide [2]. Formally, an automaton $\mathcal{A}$ is good-for-games if every two-player zero-sum game with winning condition $L(\mathcal{A})$ has the same winner as the game where the player who wins if the outcome is in $L(\mathcal{A})$ additionally has to construct a witnessing run of $\mathcal{A}$ during the play. This definition comes in two forms, depending on whether one considers only finitely branching (weak compositionality) or all games (compositionality).

Recently, the difference between being history-deterministic and both types of compositionality has been studied in detail for pushdown automata [20]. These results are very general and can easily be transferred to PA and 1-RBCM. They show that for PA, being history-deterministic, compositionality, and weak compositionality all coincide, while for 1-RBCM, being history-deterministic and compositionality coincide, but not weak compositionality.

The reason for this difference can be traced back to the fact that 1-RBCM may contain transitions that do not move the reading head (which are essentially ε-transitions), but that have side-effects beyond state changes, i.e., the counters are updated. This means that an unbounded number of configurations can be reached by processing a single letter, which implies that the game composed of an arena and a 1-RBCM may have infinite branching. So, while HDPA and 1-HDRBCM are expressively equivalent, they, perhaps surprisingly, behave differently when it comes to compositionality. We plan to investigate these differences in future work.

References

1 Marc Bagnol and Denis Kuperberg. Büchi good-for-games automata are efficiently recognizable. In Sumit Ganguly and Paritosh Pandya, editors, *FSTTCS 2018*, volume 122 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.FSTTCS.2018.16.

2 Udi Boker and Karoliina Lehtinen. History determinism vs. good for gameness in quantitative automata. In Mikołaj Bojańczy and Chandra Chekuri, editors, *FSTTCS 2021*, volume 213 of *LIPIcs*, pages 38:1–38:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSTTCS.2021.38.

3 Udi Boker and Karoliina Lehtinen. Token games and history-deterministic quantitative automata. In Patricia Bouyer and Lutz Schröder, editors, *FOSSACS 2022*, volume 13242 of *LNCS*, pages 120–139. Springer, 2022. doi:10.1007/978-3-030-99253-8_7.

4 Alin Bostan, Arnaud Carayol, Florent Koechlin, and Cyril Nicaud. Weakly-unambiguous Parikh automata and their link to holonomic series. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *ICALP 2020*, volume 168 of *LIPIcs*, pages 114:1–114:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.114.

5 Michaël Cadilhac, Alain Finkel, and Pierre McKenzie. On the expressiveness of Parikh automata and related models. In Rudolf Freund, Markus Holzer, Carlo Mereghetti, Friedrich Otto, and Beatrice Palano, editors, *NCMA 2011*, volume 282 of *books@ocg.at*, pages 103–119. Austrian Computer Society, 2011.

6 Michaël Cadilhac, Alain Finkel, and Pierre McKenzie. Unambiguous constrained automata. *Int. J. Found. Comput. Sci.*, 24(7):1099–1116, 2013. doi:10.1142/S0129054113400339.

7 Giusi Castiglione and Paolo Massazza. On a class of languages with holonomic generating functions. *Theor. Comput. Sci.*, 658:74–84, 2017. doi:10.1016/j.tcs.2016.07.022.

8 Lorenzo Clemente, Wojciech Czerwinski, Slawomir Lasota, and Charles Paperman. Regular Separability of Parikh Automata. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *ICALP 2017*, volume 80 of *LIPIcs*, pages 117:1–117:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.117.

9 Thomas Colcombet. The theory of stabilisation monoids and regular cost functions. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, *ICALP 2009, (Part II)*, volume 5556 of *LNCS*, pages 139–150. Springer, 2009. doi:10.1007/978-3-642-02930-1_12.

10 Luc Dartois, Emmanuel Filiot, and Jean-Marc Talbot. Two-way Parikh automata with a visibly pushdown stack. In Mikolaj Bojanczyk and Alex Simpson, editors, *FOSSACS 2019*, volume 11425 of *LNCS*, pages 189–206. Springer, 2019. doi:10.1007/978-3-030-17127-8_11.

11 Jürgen Dassow and Victor Mitrana. Finite automata over free groups. *Int. J. Algebra Comput.*, 10(6):725–738, 2000. doi:10.1142/S0218196700000315.

12 François Denis, Aurélien Lemay, and Alain Terlutte. Residual finite state automata. In Afonso Ferreira and Horst Reichel, editors, *STACS 2001*, volume 2010 of *LNCS*, pages 144–157. Springer, 2001.

13 Enzo Erlich, Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen, and Martin Zimmermann. History-deterministic parikh automata. *arXiv*, 2209.07745, 2022. arXiv:2209.07745.

14 Diego Figueira and Leonid Libkin. Path logics for querying graphs: Combining expressiveness and efficiency. In *LICS 2015*, pages 329–340. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.39.

15 Emmanuel Filiot, Shibashis Guha, and Nicolas Mazzocchi. Two-way Parikh automata. In Arkadev Chattopadhyay and Paul Gastin, editors, *FSTTCS 2019*, volume 150 of *LIPIcs*, pages 40:1–40:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

16 Emmanuel Filiot, Nicolas Mazzocchi, and Jean-François Raskin. A pattern logic for automata with outputs. *Int. J. Found. Comput. Sci.*, 31(6):711–748, 2020.

17 Seymour Ginsburg and Edwin H. Spanier. Semigroups, Presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966. doi:pjm/1102994974.

18 Mario Grobler, Leif Sabellek, and Sebastian Siebertz. Parikh automata on infinite words. *arXiv*, 2301.08969, 2023. doi:10.48550/arXiv.2301.08969.

19 Mario Grobler and Sebastian Siebertz. Büchi-like characterizations for parikh-recognizable omega-languages. *arxiv*, 2302.04087, 2023. doi:10.48550/arXiv.2302.04087.

20 Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen, and Martin Zimmermann. A bit of nondeterminism makes pushdown automata expressive and succinct. *arXiv*, 2105.02611, 2021. doi:10.48550/arXiv.2105.02611.

21 Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen, and Martin Zimmermann. Parikh automata over infinite words. *arXiv*, 2207.07694, 2022. doi:10.48550/arXiv.2207.07694.

22 Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen, and Martin Zimmermann. A bit of nondeterminism makes pushdown automata expressive and succinct. In Filippo Bonchi and Simon J. Puglisi, editors, *MFCS 2021*, volume 202 of *LIPIcs*, pages 53:1–53:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.MFCS.2021.53.

23 Thomas A. Henzinger, Karoliina Lehtinen, and Patrick Totzke. History-deterministic timed automata. In *CONCUR 2022*, 2022. To appear.

24 Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In Zoltán Ésik, editor, *CSL 2006*, volume 4207 of *LNCS*, pages 395–410. Springer, 2006. doi:10.1007/11874683_26.

25 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

26 Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978. doi:10.1145/322047.322058.

27 Felix Klaedtke and Harald Rueß. Monadic second-order logics with cardinalities. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *ICALP 2003*, volume 2719 of *LNCS*, pages 681–696. Springer, 2003. doi:10.1007/3-540-45061-0_54.

28 Denis Kuperberg and Michal Skrzypczak. On determinisation of good-for-games automata. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *ICALP 2015 (Part II)*, volume 9135 of *LNCS*, pages 299–310. Springer, 2015. doi:10.1007/978-3-662-47666-6_24.

29 Karoliina Lehtinen and Martin Zimmermann. Good-for-games ω-pushdown automata. *LMCS*, 18(1), 2022. doi:10.46298/lmcs-18(1:3)2022.

30 Marvin L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

31 Victor Mitrana and Ralf Stiebe. Extended finite automata over groups. *Discret. Appl. Math.*, 108(3):287–300, 2001. doi:10.1016/S0166-218X(00)00200-6.

32 Karianto Wong. Parikh automata with pushdown stack, 2004. Diploma thesis, RWTH Aachen University. URL: https://old.automata.rwth-aachen.de/download/papers/karianto/ka04.pdf.

Computing Minimal Distinguishing Hennessy-Milner Formulas is NP-Hard, but Variants are Tractable

Jan Martens
Eindhoven University of Technology, The Netherlands

Jan Friso Groote
Eindhoven University of Technology, The Netherlands

Abstract

We study the problem of computing minimal distinguishing formulas for non-bisimilar states in finite LTSs. We show that this is NP-hard if the size of the formula must be minimal. Similarly, the existence of a short distinguishing trace is NP-complete. However, we can provide polynomial algorithms, if minimality is formulated as the minimal number of nested modalities, and it can even be extended by recursively requiring a minimal number of nested negations. A prototype implementation shows that the generated formulas are much smaller than those generated by the method introduced by Cleaveland.

2012 ACM Subject Classification Theory of computation → Modal and temporal logics

Keywords and phrases Distinguishing behaviour, Hennessy-Milner logic, NP-hardness

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.32

Related Version *Full Version*: `https://arxiv.org/abs/2307.05265`

Supplementary Material *Software*: `https://github.com/jjmartens/distinguishing-hml`
archived at `swh:1:dir:c38076d88d2e9cc0bf081739203a2474ba87b7d3`

Funding AVVA project NWO 612.001.751/TOP1.17.002.

1 Introduction

Hennessy-Milner Logic (HML) [11] can be used to explain behavioural inequivalence. If two states are not bisimilar there is a *distinguishing formula* that is valid in one state but not in the other. As the reason for the states not being bisimilar can be very subtle, such a distinguishing formula is of great help to pinpoint the cause of the inequivalence.

Cleaveland [6] introduces an efficient algorithm to calculate distinguishing formulas by back-tracking the partition refinement sequence that decides bisimilarity. He states that the formulas are minimal "in a precisely defined sense". This method is used in the mCRL2 toolset [5]. However, the generated formulas are unexpectedly large. This leads to the question in which sense distinguishing formulas are minimal and how difficult it is to obtain them. Similar questions were posed throughout the literature. Some also questioned the size of the formulas – in the setting of CTL [4], others explicitly stated that they were not minimal [25, 3], and there are even suggestions that minimisation could be NP-hard [26].

In this work we answer the question by proving that in general calculating minimal distinguishing Hennessy-Milner formulas is NP-hard. Minimality can be taken rather broadly, as having a minimal number of symbols, modalities, or logical connectives. As observed in [8] a distinguishing formula can be exponential in size. However, as was already noted in [6], when using sharing in the representation of formulas, for instance by formulating the distinguishing formula as a set of equations or a directed acyclic graph, the representation is polynomial. Calculating a minimal shared distinguishing formula is NP-complete.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 32; pp. 32:1–32:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The proof of this result uses a reduction directly from CNF-SAT and the construction is similar to the construction used by Hunt [13] where it is shown that deciding equivalence of acyclic non-deterministic automata is NP-complete. We show via the NP-hardness of deciding whether there is a distinguishing trace for an acyclic non-deterministic LTS, that computing minimal HML formulas is also NP-hard.

As distinguishing formulas are very useful, we are wondering whether a variant of minimality of distinguishing formulas exists that leads to concise formulas and that can effectively be calculated. We answer this positively by providing efficient algorithms to construct distinguishing formulas that are minimal with respect to the *observation-depth*, i.e., the number of nested modalities. Within this we can even guarantee in polynomial time that the *negation-depth*, i.e., the number of nested negations, or equivalently the number of nested alternations of box and diamond modalities, is minimal. These algorithms strictly improve upon the method by Cleaveland [6]. A prototype implementation of our algorithm shows that our formulas are indeed much smaller and more pleasant to use. In order to obtain these results we employ the notions of k-bisimilarity [19] and m-nested similarity [10].

Distinguishing formulas have been the topic of studies in many papers, more than we can mention. A recent impressive work introduces a method to find minimal distinguishing formulas for various classes of behavioural equivalences [3]. The algorithm translates the problem to determining the winning region in a reachability game. These games can grow super-exponentially in size. In the context of distinguishing deterministic finite automata, an algorithm is given that from a splitting tree finds pairwise minimal distinguishing words [23]. In a more generalized setting [25, 15] a co-algebraic method is given to generate distinguishing modal formulas. The notion of distinguishing formulas is also used in the setting with abstractions for branching bisimilarity [16, 9].

This document is structured as follows. In Section 2 the required preliminaries on LTSs and HML formulas are given. In Section 3, we show that decision problems related to finding minimal distinguishing formulas are NP-hard. Next, in Section 4 we give a procedure that generates a minimal observation- and negation-depth formula. Additionally, in this section, we give a partition refinement algorithm inspired by [23, 20] which can be used to determine minimal observation-depth distinguishing formulas. In the full version, an appendix is included containing proofs omitted here due to space constraints.

2 Preliminaries

For the numbers $i, j \in \mathbb{N}$, we define $[i, j] = \{c \in \mathbb{N} \mid i \leqslant c \leqslant j\}$, the closed interval from i to j.

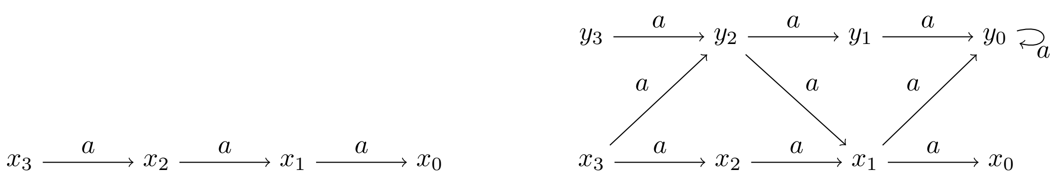
2.1 LTSs, k-bisimilarity & m-nested similarity

We use Labelled Transition Systems (LTSs) as our behavioural models. Strong bisimilarity is a widely used behavioural equivalence [19, 22], which we define in the classical inductive way.

▶ **Definition 1.** *A labelled transition system (LTS) $L = (S, Act, \rightarrow)$ is a three-tuple containing:*

- *a finite set of states S,*
- *a finite set of action labels Act, and*
- *a transition relation $\rightarrow \subseteq S \times Act \times S$.*

We write $s \xrightarrow{a} s'$ iff $(s, a, s') \in \rightarrow$. We call s' an a-derivative of s iff $s \xrightarrow{a} s'$.



(a) The LTS $\mathcal{A}_3 = (\{x_0, \ldots, x_3\}, \{a\}, \rightarrow)$.

(b) The LTS $\mathcal{B}_3 = (\{x_i, y_i \mid 0 \leqslant i \leqslant 3\}, \{a\}, \rightarrow)$.

Figure 1 Two example LTSs.

▶ **Definition 2** (k-bisimilar [19]). *Let $L = (S, Act, \rightarrow)$ be an LTS. For every $k \in \mathbb{N}$, k-bisimilarity written as $\mathrel{\underline{\leftrightarrow}}_k$ is defined inductively:*

$$\begin{aligned} \mathrel{\underline{\leftrightarrow}}_0 &= \{(s,t) \mid s, t \in S\}, \text{ and} \\ \mathrel{\underline{\leftrightarrow}}_{k+1} &= \{(s,t) \mid \forall s \xrightarrow{a} s'. \exists t \xrightarrow{a} t' \text{ such that } s' \mathrel{\underline{\leftrightarrow}}_k t', \text{ and} \\ &\qquad \forall t \xrightarrow{a} t'. \exists s \xrightarrow{a} s' \text{ such that } t' \mathrel{\underline{\leftrightarrow}}_k s'\}. \end{aligned}$$

Bisimilarity, denoted as $\mathrel{\underline{\leftrightarrow}}$, is defined as the intersection of all k-bisimilarity relations for all $k \in \mathbb{N}$: $\mathrel{\underline{\leftrightarrow}} = \bigcap_{k \in \mathbb{N}} \mathrel{\underline{\leftrightarrow}}_k$. As our transition systems are finite, and therefore finitely branching, $\mathrel{\underline{\leftrightarrow}}$ coincides with the more general co-inductive definition of bisimulation [22]. The intuition behind $\mathrel{\underline{\leftrightarrow}}_i$ is that within i (atomic) observations there is no distinguishing behaviour. We sketch a rather simple example that showcases this behaviour.

▶ **Example 3.** For every $n \in \mathbb{N}$, we define the LTS $\mathcal{A}_n = (S, \{a\}, \rightarrow)$ with a singleton action set, and the set of states $S = \{x_0, \ldots, x_n\}$. The transition function contains a single path $x_i \xrightarrow{a} x_{i-1}$ for all $1 \leqslant i \leqslant n$.

In Figure 1a the LTS $\mathcal{A}_3$ is shown. A state x_i can perform i a-transitions ending in a deadlock state. All states in $\mathcal{A}_3$ are behaviourally inequivalent. Intuitively, we see that distinguishing the states x_3 and x_2 takes at least 3 observations.

In general, it holds that for $n \in \mathbb{N}$, the states x_n and x_{n-1} of the LTS $\mathcal{A}_n$ are $n-1$-bisimilar but not n-bisimilar, i.e. $x_n \mathrel{\underline{\leftrightarrow}}_{n-1} x_{n-1}$ but $x_n \not\mathrel{\underline{\leftrightarrow}}_n x_{n-1}$. In order to distinguish these states we require n (atomic) observations. This intuition is formalized in Theorem 10.

▶ **Fact 4.** *We state these well-known facts for an LTS $L = (S, Act, \rightarrow)$, and $k \in \mathbb{N}$:*

1. *The relation $\mathrel{\underline{\leftrightarrow}}_k$ is an equivalence relation.*
2. *If two states are k-bisimilar, they are l-bisimilar for every $l \leqslant k$.*
3. *If $\mathrel{\underline{\leftrightarrow}}_k = \mathrel{\underline{\leftrightarrow}}_{k+1}$ then $\mathrel{\underline{\leftrightarrow}}_k = \mathrel{\underline{\leftrightarrow}}_{k+u} = \mathrel{\underline{\leftrightarrow}}$, for all $u \in \mathbb{N}$.*

For technical reasons we also define m-nested similarity [10] which uses the concept of similarity.

▶ **Definition 5** (Similarity). *Given an $L = (S, Act, \rightarrow)$, we define similarity $\mathrel{\underline{\rightarrow}} \subseteq S \times S$ as the largest relation such that if $s \mathrel{\underline{\rightarrow}} t$ then for all transitions $s \xrightarrow{a} s'$ there is a $t \xrightarrow{a} t'$ such that $s' \mathrel{\underline{\rightarrow}} t'$.*

We say a state s is *simulated* by t iff $s \mathrel{\underline{\rightarrow}} t$.

▶ **Definition 6** (cf. Def. 8.5.2. [10]). *Let $L = (S, Act, \rightarrow)$ be an LTS, and $m \in \mathbb{N}$ a number. We inductively define m-nested similarity inclusion as follows: $\mathrel{\underline{\rightarrow}}^0 = \mathrel{\underline{\rightarrow}}$, and for every $i \in \mathbb{N}$, the relation $\mathrel{\underline{\rightarrow}}^{i+1} \subseteq S \times S$ is the largest relation such that for all $(s,t) \in \mathrel{\underline{\rightarrow}}^{i+1}$ it holds that:*

- *$s \mathrel{\underline{\rightarrow}}^i t$ and $t \mathrel{\underline{\rightarrow}}^i s$, and*
- *if $s \xrightarrow{a} s'$ then there is a $t \xrightarrow{a} t'$ such that $s' \mathrel{\underline{\rightarrow}}^{i+1} t'$.*

We write $\rightleftarrows^m$ as the symmetric closure of m-nested similarity inclusion, i.e. $\rightleftarrows^m = \rightrightarrows^m \cap (\rightrightarrows^m)^{-1}$, which we call *m-nested similarity.* Note that we deviate slightly from the definition in [10], where 1-nested simulation equivalence coincides with simulation equivalence.

▶ **Example 7.** For every $n \in \mathbb{N}$, we define the LTS $\mathcal{B}_n = (S, \{a\}, \rightarrow)$ with a singleton action set, the set of states $S = \{x_0, \ldots, x_n, y_0, \ldots, y_n\}$, and the transition relation containing the transition $y_0 \xrightarrow{a} y_0$ and, for every $i \in [1, n]$, the transitions:

- $y_i \xrightarrow{a} y_{i-1}$ and $x_i \xrightarrow{a} x_{i-1}$, and
- $y_i \xrightarrow{a} x_{i-1}$ if i is even, or $x_i \xrightarrow{a} y_{i-1}$ if i is odd.

In Figure 1b the LTS $\mathcal{B}_3$ is shown. We observe that x_0 is simulated by y_0, since x_0 has no outgoing transitions. So it is the case that $x_0 \rightrightarrows^0 y_0$, but $y_0 \not\rightrightarrows^0 x_0$, and hence $x_0 \not\rightleftarrows^0 y_0$. In general, for all $n \geq 1$ it holds in the LTS $\mathcal{B}_n$ that $x_n \rightleftarrows^{n-1} y_n$, but $x_n \not\rightleftarrows^n y_n$.

2.2 Hennessy-Milner logic (HML)

We use Hennessy-Milner Logic (HML) [11] to distinguish states. For some finite set of actions *Act*, the syntax of HML is defined as

$$\phi ::= tt \mid \langle a \rangle \phi \mid \neg \phi \mid \phi \wedge \phi,$$

where $a \in Act$. The logic consists of three necessary elements:

- *Observations* $\langle a \rangle \phi$, the state witnesses an observation a to a state that satisfies ϕ.
- *Negations* $\neg \phi$, the state does not satisfy ϕ.
- *Conjunctions* $\phi_1 \wedge \phi_2$, the state satisfies both ϕ_1 and ϕ_2.

The set $\mathcal{F}$ is defined to contain all HML formulas. It is common to use the abbreviations $ff = \neg tt$, $[a]\phi = \neg \langle a \rangle \neg \phi$ and $\phi_1 \vee \phi_2 = \neg(\neg \phi_1 \wedge \neg \phi_2)$.

Given an LTS $L = (S, Act, \rightarrow)$, we define the semantics of this logic $[\![-]\!]_L : \mathcal{F} \rightarrow 2^S$, inductively as follows:

$$\begin{aligned}
[\![tt]\!]_L &= S, \\
[\![\langle a \rangle \phi]\!]_L &= \{s \in S \mid \exists s' \in S \text{ s.t. } s \xrightarrow{a} s' \text{ and } s' \in [\![\phi]\!]_L\}, \\
[\![\neg \phi]\!]_L &= S \setminus [\![\phi]\!]_L, \text{ and} \\
[\![\phi_1 \wedge \phi_2]\!]_L &= [\![\phi_1]\!]_L \cap [\![\phi_2]\!]_L,
\end{aligned}$$

for $a \in Act$ and $\phi, \phi_1, \phi_2 \in \mathcal{F}$. This function yields for a formula $\phi \in \mathcal{F}$ the subset of S where ϕ is true. Often we omit the reference to the LTS L when it is clear from the context.

We use HML formulas to describe distinguishing behaviour. Let $L = (S, Act, \rightarrow)$ be an LTS, $s \in S$ and $t \in S$ states, and $\phi \in \mathcal{F}$ a HML formula. We write $s \sim_\phi t$ iff $s \in [\![\phi]\!] \Leftrightarrow t \in [\![\phi]\!]$, and conversely $s \not\sim_\phi t$ iff $s \in [\![\phi]\!] \Leftrightarrow t \notin [\![\phi]\!]$. Additionally, we write $s \leqslant_\phi t$ if $s \in [\![\phi]\!] \Rightarrow t \in [\![\phi]\!]$. Given a set of HML formulas $\mathcal{G}$ we write $s \sim_\mathcal{G} t$ iff for every $\psi \in \mathcal{G}$, it holds that $s \sim_\psi t$. Similarly, we write $s \leqslant_\mathcal{G} t$ iff $s \leqslant_\psi t$ for all $\psi \in \mathcal{G}$.

▶ **Definition 8.** *Given an LTS $L = (S, Act, \rightarrow)$ and two states $s, t \in S$, then a formula $\phi \in \mathcal{F}$ distinguishes s and t iff $s \not\sim_\phi t$.*

2.2.1 Metrics

To express the size of a formula we use three different metrics:

- *size* the total number of observations,
- *observation-depth* the largest number of nested observation in the formula, and

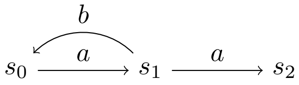


Figure 2 The LTS $M = (S, Act, \to)$, $Act = \{a, b\}$ and $S = \{s_0, s_1, s_2\}$.

- *negation-depth* the largest number of nested negations in the formula.

For these metrics we inductively define the functions $|\cdot| : \mathcal{F} \to \mathbb{N}$ for size, $d_\diamond : \mathcal{F} \to \mathbb{N}$ for observation-depth and $d_\neg : \mathcal{F} \to \mathbb{N}$ for negation-depth, as follows:

$$\begin{array}{llllll}
|tt| & = 0, & d_\diamond(tt) & = 0, & d_\neg(tt) & = 0, \\
|\langle a\rangle\phi| & = |\phi| + 1, & d_\diamond(\langle a\rangle\phi) & = d_\diamond(\phi) + 1, & d_\neg(\langle a\rangle\phi) & = d_\neg(\phi), \\
|\neg\phi| & = |\phi|, & d_\diamond(\neg\phi) & = d_\diamond(\phi), & d_\neg(\neg\phi) & = d_\neg(\phi) + 1, \\
|\phi_1 \wedge \phi_2| & = |\phi_1| + |\phi_2|. & d_\diamond(\phi_1 \wedge \phi_2) & = \max(d_\diamond(\phi_1), d_\diamond(\phi_2)). & d_\neg(\phi_1 \wedge \phi_2) & = \max(d_\neg(\phi_1), d_\neg(\phi_2)).
\end{array}$$

Given natural numbers $n, m \in \mathbb{N}$ we define the sets $\mathcal{F}_n$ and $\mathcal{F}^m$ as the fragment of HML formulas with bounded observation- and respectively negation-depth, i.e. $\mathcal{F}_n = \{\phi \mid d_\diamond(\phi) \leqslant n\}$, and $\mathcal{F}^m = \{\phi \mid d_\neg(\phi) \leqslant m\}$.

We write $\mathcal{F}_n^m$ for the set $\mathcal{F}_n^m = \mathcal{F}_n \cap \mathcal{F}^m$. Based on these metrics we define multiple notions of *minimal* distinguishing formulas.

▶ **Definition 9.** *Given an LTS $L = (S, Act, \to)$, let $\phi \in \mathcal{F}$ be an HML formula that distinguishes $s \in S$ and $t \in S$. Then in distinguishing s and t, the formula ϕ is called:*

- *to have* minimal observation-depth *iff ϕ has the least nested modalities, i.e. for all $\phi' \in \mathcal{F}$ if $s \not\sim_{\phi'} t$ then $d_\diamond(\phi) \leqslant d_\diamond(\phi')$;*
- *to have* minimal negation-depth *iff ϕ has the least nested negations, i.e., for all $\phi' \in \mathcal{F}$ if $s \not\sim_{\phi'} t$ then $d_\neg(\phi) \leqslant d_\neg(\phi')$;*
- *to be* minimal *iff ϕ has the least number of modalities, i.e., for all $\phi' \in \mathcal{F}$ if $s \not\sim_{\phi'} t$ then $|\phi| \leqslant |\phi'|$;*
- *to have* minimal observation- and negation-depth *iff it is minimal in the lexicographical order of observation and negation-depth, i.e., iff for all $\phi' \in \mathcal{F}$ if $s \not\sim_{\phi'} t$ then $d_\diamond(\phi) \leqslant d_\diamond(\phi')$ and if $d_\diamond(\phi) = d_\diamond(\phi')$ then $d_\neg(\phi) \leqslant d_\neg(\phi')$;*
- irreducible *[6, Def. 2.5] iff no ϕ' obtained by replacing a non-trivial subformula of ϕ with the formula tt distinguishes s from t.*

The first three notions correspond directly to the metrics we defined. The notion of *irreducible* distinguishing formulas corresponds to the minimality notion used in the work by Cleaveland [6]. The different notions are not comparable. This is witnessed by the LTS M pictured in Figure 2. The formula $\phi_1 = \langle a\rangle\langle a\rangle tt$ distinguishes s_0 and s_1 since $s_0 \in [\![\phi_1]\!]$ and $s_1 \notin [\![\phi_1]\!]$. Additionally, ϕ_1 is irreducible, since any formula obtained by replacing a subformula by tt is not a distinguishing formula. However, the formula ϕ_1 is not *minimal* since the formula $\phi_2 = \langle b\rangle tt$ also distinguishes s_0 and s_1.

2.2.2 Representation

A note has to be made on the representation of distinguishing formulas. It is known that distinguishing formulas can grow very large. In fact there is a family of LTSs that showcases an exponential lower bound on the size of the minimal distinguishing formula [8, 25]. This exponential lower bound is not in contradiction with the polynomial-time algorithm from Cleaveland [6] since [6] uses equations to represent the subformulas. For example the formula $\langle a\rangle\langle b\rangle\langle c\rangle tt \wedge \langle b\rangle\langle c\rangle tt$ can be represented using the equations $\phi_1 = \langle a\rangle\phi_2 \wedge \phi_2$ and $\phi_2 = \langle b\rangle\langle c\rangle tt$, or as the term in Figure 3.

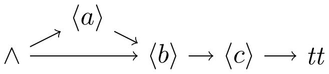


Figure 3 A HML formula represented as a shared term.

The shared representation does not change the observation-depth and the negation-depth. The size of a formula is influenced, but it does not affect the NP-hardness result.

2.2.3 Correspondences

There are strong correspondences between different fragments of HML on the one hand and m-nested similarity and bisimilarity on the other hand. We use these to obtain minimal distinguishing formulas. The first theorem states that those HML formulas that have at most k-nested observations exactly capture k-bisimilarity.

▶ **Theorem 10** ((cf. [11, Theorem 2.2])). *Given an LTS $L = (S, Act, \rightarrow)$ and two states $s, t \in S$. For every $k \in \mathbb{N}$,*

$$s \mathrel{\underline{\leftrightarrow}}_k t \iff s \sim_{\mathcal{F}_k} t.$$

In this work we are mainly interested in the contraposition of this theorem. For every $k \in \mathbb{N}$, two states $s, t \in S$ are not k-bisimilar iff there is a $\phi \in \mathcal{F}_k$ that distinguishes s and t, i.e. $s \not\sim_\phi t$. For this reason for every $k \in \mathbb{N}$ we call s and t *k-distinguishable* iff $s \not\mathrel{\underline{\leftrightarrow}}_k t$. We call the states s and t *distinguishable* iff they are k-distinguishable for some $k \in \mathbb{N}$.

▶ **Corollary 11.** *Given an LTS $L = (S, Act, \rightarrow)$ and two states $s, t \in S$. For every $k \in \mathbb{N}$,*

$$s \not\mathrel{\underline{\leftrightarrow}}_k t \iff \textit{there is a formula } \phi \in \mathcal{F}_k \textit{ such that } s \not\sim_\phi t.$$

In [10] it is shown that fragments of HML with bounded negation-depth allow a similar relational classification. The following theorem relates the fragment $\mathcal{F}^m$ to m-nested similarity inclusion.

▶ **Theorem 12** ((cf. [10, Corollary 8.7.6])). *Let $L = (S, Act, \rightarrow)$ be an LTS, then for all $m \in \mathbb{N}$, and states $s, t \in S$:*

$$s \mathrel{\underline{\rightarrow}}^m t \iff s \leqslant_{\mathcal{F}^m} t.$$

The main use for our work is that if two states are not m-nested similar, then there is a distinguishing formula with at most m nested negations.

▶ **Corollary 13.** *Let $L = (S, Act, \rightarrow)$ be an LTS, then for all $m \in \mathbb{N}$, and states $s, t \in S$:*

$$s \not\mathrel{\underline{\rightarrow}}^m t \iff \textit{there is a formula } \phi \in \mathcal{F}^m \textit{ s.t. } s \in [\![\phi]\!] \textit{ and } t \notin [\![\phi]\!].$$

Let us recall the LTS $\mathcal{A}_3$ from Example 3 drawn in Figure 1a. In this LTS we see that $x_3 \mathrel{\underline{\leftrightarrow}}_2 x_2$, but $x_3 \not\mathrel{\underline{\leftrightarrow}}_3 x_2$. As a result of Corollary 11 we know that there is a formula $\phi \in \mathcal{F}_3$ that distinguishes x_3 and x_2. This is witnessed by the formula $\phi = \langle a\rangle\langle a\rangle\langle a\rangle tt \in \mathcal{F}_3$, which is a distinguishing formula, since $x_3 \in [\![\phi]\!]$ and $x_2 \notin [\![\phi]\!]$. We also see that $x_3 \sim_{\mathcal{F}_2} x_2$, hence there is no such formula in $\mathcal{F}_2$.

For the LTS $\mathcal{B}_3$ from Example 7, we aim to distinguish the states x_3 and y_3. According Corollary 13 there is a distinguishing formula $\phi \in \mathcal{F}^3$, since $x_3 \not\mathrel{\underline{\rightarrow}}^3 y_3$. This is witnessed by the formula $\phi = \langle a\rangle\neg\langle a\rangle\neg\langle a\rangle\neg\langle a\rangle tt$. This is a distinguishing formula as $x_3 \in [\![\phi]\!]$ and $y_3 \notin [\![\phi]\!]$. Corollary 13 also shows that this is the minimal negation-depth formula distinguishing x_3 and y_3, as $x_3 \rightleftarrows^2 y_3$.

2.2.4 Traces

Let Act be a finite set of action labels. We denote by $Act^* := \bigcup_{i \in \mathbb{N}} Act^i$ the set of all finite sequences on the action labels Act. We write ε for the empty sequence. For sequences $w, u \in Act^*$, we denote with $|w|$ its length and $w \cdot u$ the concatenation of w and u, which is sometimes also written as wu.

▶ **Definition 14.** *Given an LTS $L = (S, Act, \rightarrow)$. The set of* traces *$Tr(s) \subseteq Act^*$ of a state $s \in S$ is the smallest set satisfying:*

1. *$\varepsilon \in Tr(s)$, and*
2. *for an action $a \in Act$, and state $s' \in S$ if a trace $w \in Tr(s')$ and $s \xrightarrow{a} s'$, then $aw \in Tr(s)$.*

Inductively, we define the formula ϕ_w for every word $w \in Act^*$, such that $\phi_\varepsilon = tt$, and $\phi_{aw} = \langle a \rangle \phi_w$. We call a formula $\phi \in \mathcal{F}$ a *trace-formula* iff there is a sequence $w \in Act^*$ such that $\phi = \phi_w$.

▶ **Lemma 15.** *Let $L = (S, Act, \rightarrow)$ be an LTS, and $w \in Act^*$ a trace. Then for all $s \in S$:*

$$s \in [\![\phi_w]\!] \iff w \in Tr(s).$$

Two states $s \in S$ and $t \in S$ in an LTS $L = (S, Act, \rightarrow)$ are said to be trace-equivalent iff $Tr(s) = Tr(t)$. Bisimilarity is a more fine-grained equivalence than trace equivalence. Two states $s \in S$ and $t \in S$ can be trace-equivalent, while not being bisimilar. In this case there is a formula $\phi \in F$ such that $s \not\sim_\phi t$ and we know that ϕ is not a trace-formula. However, ϕ contains traces that are both traces of s and t. To make this more precise we define the traces of a formula by induction for formulas $\phi, \phi_1, \phi_2 \in \mathcal{F}$ as follows:

$$\begin{aligned} Tr(tt) &= \{\varepsilon\}, \\ Tr(\langle a \rangle \phi) &= \{a\} \cup \{a \cdot w \mid w \in Tr(\phi)\}, \\ Tr(\neg \phi) &= Tr(\phi), \\ Tr(\phi_1 \wedge \phi_2) &= Tr(\phi_1) \cup Tr(\phi_2). \end{aligned}$$

The traces of a formula allow us to state the correspondence between k-distinguishability and the length of shared traces. We formulate this using the minimal observation depth that, given two distinguishable states, yields the smallest $i \in \mathbb{N}$ such that the states are i-distinguishable:

▶ **Definition 16.** *Let $L = (S, Act, \rightarrow)$ be an LTS. We define the minimal observation depth $\Delta : S \times S \rightarrow \mathbb{N} \cup \{\infty\}$ by*

$$\Delta(s,t) = \begin{cases} i & \text{if } s \not\underline{\leftrightarrow}_i t, \text{ and } s \underline{\leftrightarrow}_{i-1} t, \\ \infty & \text{if } s \underline{\leftrightarrow} t. \end{cases}$$

The next lemma says that if states have minimal observation depth i, then any distinguishing formula contains a trace of length at least i.

▶ **Lemma 17.** *Let $L = (S, Act, \rightarrow)$ be an LTS and $s, t \in S$ two distinguishable states such that $\Delta(s,t) = i$ for some $i \in \mathbb{N}$. For all $\phi \in \mathcal{F}$, if $s \not\sim_\phi t$ then there is a trace $w \in Tr(\phi)$ such that $|w| \geq i$ and $w \in Tr(s) \cup Tr(t)$.*

Proof sketch. Proven by induction on the shape of ϕ. The only interesting case is if $\phi = \langle a \rangle \phi'$ for some $a \in Act$ and $\phi' \in \mathcal{F}$. Assume without loss of generality that $s \in [\![\phi]\!]$ and $t \notin [\![\phi]\!]$. This means that there is a transition $s \xrightarrow{a} s'$ such that $s' \in [\![\phi']\!]$. Since $\Delta(s,t) = i$ there is also a $t \xrightarrow{a} t'$ such that $\Delta(s', t') = i - 1$.

Since $t \notin [\![\phi]\!]$ also $t' \notin [\![\phi']\!]$, and thus we can apply our induction hypothesis to conclude that there is a $w' \in \mathit{Tr}(\phi')$ such that $|w'| \geq i-1$ and $w' \in \mathit{Tr}(s') \cup \mathit{Tr}(t')$. From w' we construct aw' and observe that $aw' \in \mathit{Tr}(\phi)$, $aw' \in \mathit{Tr}(s) \cup \mathit{Tr}(t)$ and $|aw'| \geq i$, which finishes the proof. ◀

3 NP-hardness results

In this section we show that finding minimal distinguishing formulas is NP-hard.

We first show that the existence of a short trace is NP-complete similar to a result of Hunt [13, Sec. 2.2] on acyclic NFAs. A corollary of the construction is that finding the *minimal size* distinguishing formula is NP-hard.

We define the decision problems *TRACE-DIST* and *MIN-DIST*. Given an LTS $L = (S, \mathit{Act}, \rightarrow)$, two states $s, t \in S$ such that $s \not\approx_i t$ for $i = |S|$, and a number $l \in \mathbb{N}$.
TRACE-DIST: There is a trace-formula $\phi \in \mathcal{F}_i$, such that ϕ distinguishes s and t.
MIN-DIST: There is a formula $\phi \in \mathcal{F}_i$, such that ϕ distinguishes s and t, and $|\phi| \leqslant l$.
We point out that *TRACE-DIST* is not the same as deciding trace-equivalence. The problem *TRACE-DIST* decides whether there is a distinguishing trace of length i, and i is smaller than the number of states, and a minimal distinguishing trace might be super-polynomial in size [7, Sec. 5].

3.1 Reduction

We prove that *TRACE-DIST* is NP-complete and *MIN-DIST* is NP-hard by a reduction from the decision problem *CNF-SAT*. This decision problem decides whether a given propositional formula $\mathcal{C}$ in conjunctive normal form (CNF) is satisfiable. For this we define an LTS $L_{\mathcal{C}}$, based on the CNF formula $\mathcal{C}$.

▶ **Definition 18.** *Let $\mathcal{C} = C_1 \wedge \ldots \wedge C_n$ be a CNF formula over the set of proposition letters $\mathit{Prop} = \{p_1, \ldots, p_k\}$. We define the LTS $L_{\mathcal{C}} = (S, \mathit{Act}, \rightarrow)$ as follows:*

- *The set of states S is defined as*

$$\begin{aligned} S = &\{\mathtt{unsat}_i^C \mid C \in \{C_1, \ldots, C_n\}, i \in [0,k]\} \cup \{\mathtt{sat}_i \mid i \in [0,k]\} \\ &\cup \{\bot_i \mid i \in [0,k]\} \cup \{s, t, \delta\}. \end{aligned}$$

- *The set of actions Act is defined as*

$$\mathit{Act} = \{p, \overline{p} \mid p \in \mathit{Prop}\} \cup \{\mathtt{init}, \mathtt{false}\}.$$

- *The relation $\rightarrow$ contains for each $C \in \{C_1, \ldots, C_n\}$ and $i \in [1,k]$:*

$$\begin{aligned} \mathtt{unsat}_{i-1}^C &\xrightarrow{p_i} \begin{cases} \mathtt{sat}_i & \text{if } p_i \text{ is a literal of } C, \\ \mathtt{unsat}_i^C & \text{otherwise,} \end{cases} \\ \mathtt{unsat}_{i-1}^C &\xrightarrow{\overline{p_i}} \begin{cases} \mathtt{sat}_i & \text{if } \neg p_i \text{ is a literal of } C, \\ \mathtt{unsat}_i^C & \text{otherwise,} \end{cases} \\ \mathtt{sat}_{i-1} &\xrightarrow{x} \mathtt{sat}_i \text{ for } x \in \{p_i, \overline{p_i}\}, \text{ and} \\ \bot_{i-1} &\xrightarrow{x} \bot_i \text{ for } x \in \{p_i, \overline{p_i}\}. \end{aligned}$$

Additionally, it contains the auxiliary transitions

$$\begin{aligned} \mathtt{unsat}_k^C &\xrightarrow{\mathtt{false}} \delta \text{ for } C \in \{C_1, \ldots, C_n\}, \\ \bot_k &\xrightarrow{\mathtt{false}} \delta, \\ t &\xrightarrow{\mathtt{init}} \mathtt{unsat}_0^C \text{ for } C \in \{C_1, \ldots, C_n\}, \\ t &\xrightarrow{\mathtt{init}} \mathtt{sat}_0, \\ s &\xrightarrow{\mathtt{init}} \mathtt{sat}_0, \text{ and} \\ s &\xrightarrow{\mathtt{init}} \bot_0. \end{aligned}$$

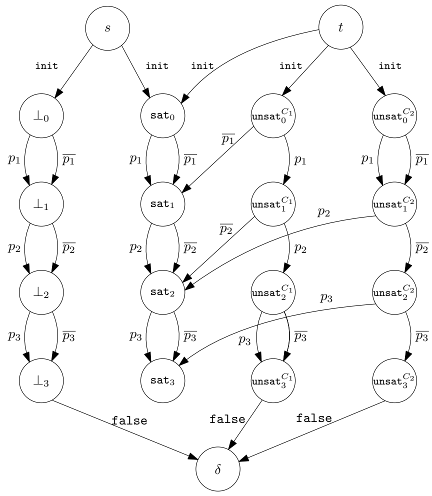


Figure 4 The LTS $L_{\mathcal{C}}$ for the formula $\mathcal{C} = (\neg p_1 \vee \neg p_2) \wedge (p_2 \vee p_3)$.

The LTS $L_{\mathcal{C}}$ for the CNF formula $\mathcal{C} = C_1 \wedge C_2$ with clauses $C_1 = \neg p_1 \vee \neg p_2$ and $C_2 = p_2 \vee p_3$ is depicted in Figure 4.

In this construction an interpretation of the propositions $Prop = \{p_1, \ldots, p_k\}$ is directly related to a word $w = a_1 \ldots a_k$, where $a_i \in \{p_i, \overline{p_i}\}$ for every $i \in [1, k]$. The set of truth assignments encoded as words is defined as:

$$\mathit{Truths} = \{a_1 \ldots a_k \mid a_i \in \{p_i, \overline{p_i}\} \text{ for all } i \in [1, k]\}.$$

Given a truth assignment $\rho : Prop \to \mathbb{B}$, we define w_ρ as $w_\rho = a_1 \ldots a_k$, where $a_i = p_i$ if $\rho(p_i) = \mathit{true}$ and $a_i = \overline{p_i}$, otherwise. Conversely, for a word $w = a_1 \ldots a_k$, a trace from *Truths*, it represents the truth assignment ρ_w defined for each $i \in [1, k]$ as:

$$\rho_w(p_i) = \begin{cases} \mathit{true} & \text{if } a_i = p_i, \\ \mathit{false} & \text{if } a_i = \overline{p}_i. \end{cases}$$

The idea of the construction of $L_{\mathcal{C}}$ is that it contains a $\bot$ component, a $\mathtt{sat}$ component, and an $\mathtt{unsat}^C$ component for every clause C. All components are deterministic and acyclic, and hence describe a finite set of traces. All the traces of these components start by a truth assignment $w \in \mathit{Truths}$. By construction, for every truth assignment $w \in \mathit{Truths}$, $w \cdot \mathtt{false} \in \mathit{Tr}(\bot_0)$. In this way the $\bot$ component represents falsehood. Conversely, the state $\mathtt{sat}_0$ represents a tautology, since for any truth assignment $w \in \mathit{Truths}$, $w \cdot \mathtt{false} \notin \mathit{Tr}(\mathtt{sat}_0)$. For every clause C, and truth assignment $w \in \mathit{Truths}$ the state $\mathtt{unsat}_0^C$ contains $w \cdot \mathtt{false}$ as trace iff ρ_w does not satisfy C.

▶ **Lemma 19.** *Let $L_{\mathcal{C}} = (S, Act, \to)$ be the LTS for a CNF formula $\mathcal{C} = C_1 \wedge \ldots \wedge C_n$ with propositions $\{p_1, \ldots, p_k\}$, then:*

$$\begin{aligned} \mathit{Tr}(\mathtt{sat}_0) &= \{u \in Act^* \mid \exists w \in \mathit{Truths}.\ u \textit{ is a prefix of } w\}, \\ \mathit{Tr}(\bot_0) &= \mathit{Tr}(\mathtt{sat}_0) \cup \{w \cdot \mathtt{false} \mid w \in \mathit{Truths}\}, \textit{ and} \\ \mathit{Tr}(\mathtt{unsat}_0^C) &= \mathit{Tr}(\mathtt{sat}_0) \cup \{w \cdot \mathtt{false} \mid w \in \mathit{Truths} \textit{ and } \rho_w \textit{ does not satisfy } C\}. \end{aligned}$$

This lemma is easily verified from the construction of $L_{\mathcal{C}}$.

▶ **Corollary 20.** *Let $w \in Truths$ be a trace, and $L_{\mathcal{C}}$ the LTS for the CNF formula $\mathcal{C} = C_1 \wedge \ldots \wedge C_n$. Then for any clause $C \in \{C_1, \ldots, C_n\}$:*

$$w{\cdot}\mathtt{false} \in Tr(\mathtt{unsat}_0^C) \iff C \text{ is not satisfied under } \rho_w.$$

The following lemma contains the main idea for the reduction of the main theorem showing *TRACE-DIST* is NP-complete.

▶ **Lemma 21.** *Given the LTS $L_{\mathcal{C}} = (S, \to, Act)$ for a CNF formula $\mathcal{C} = C_1 \wedge \ldots \wedge C_n$, with propositions $Prop = \{p_1, \ldots, p_k\}$. Then there is a trace $w \in Act^{k+1}$ such that $w \in Tr(\bot_0)$, and $w \notin Tr(\mathtt{unsat}_0^C)$ for every $C \in \{C_1, \ldots, C_n\}$ if and only if $\mathcal{C}$ is satisfiable.*

Proof. We prove this in both directions separately.

($\Rightarrow$) As a witness, we obtain a trace $w \in Tr(\bot_0)$ of length at most $k{+}1$ such that $w \notin Tr(\mathtt{unsat}_0^C)$ for all clauses $C \in \{C_1, \ldots, C_n\}$. Since $w \in Tr(\bot_0)$ by Lemma 19 either $w \in Tr(\mathtt{sat}_0)$ or $w \in \{v \cdot \mathtt{false} \mid v \in Truths\}$. Since $Tr(\mathtt{sat}_0) \subseteq Tr(\mathtt{unsat}_0^C)$, and $w \notin Tr(\mathtt{unsat}_0^C)$, there is a trace $v \in Truths$ such that $w = v \cdot \mathtt{false}$. By Corollary 20 all clauses C are satisfied by ρ_w. This means ρ_w is a satisfying assignment for $\mathcal{C}$.

($\Leftarrow$) If there is a satisfying assignment ρ for $\mathcal{C}$ then we show that $w_\rho \cdot \mathtt{false}$ witnesses the implication. First observe that by definition $w_\rho \cdot \mathtt{false} \in Tr(\bot_0)$. Let $C \in \{C_1, \ldots, C_n\}$ be any clause. Since ρ is a satisfying assignment, C is satisfied under ρ. This means by Corollary 20 that $w_\rho \notin Tr(\mathtt{unsat}_0^C)$. ◀

Now we are ready to prove the main theorem of this section.

▶ **Theorem 22.** *Deciding TRACE-DIST is NP-complete.*

Proof. First we verify that TRACE-DIST is in NP. Given an LTS $L = (S, Act, \to)$, and two states $s, t \in S$. As a witness we get a formula $\phi \in \mathcal{F}_{|S|}$, which is a trace-formula. Since $d_\Diamond(\phi) \leqslant |S|$ this is polynomial in size. It is well known that given a formula ϕ we can check in polynomial time whether $s \sim_\phi t$.

To show TRACE-DIST is NP-hard we reduce CNF-SAT to TRACE-DIST. Let $\mathcal{C} = C_1 \wedge \ldots \wedge C_n$ be a CNF formula over the propositions $Prop = \{p_1, \ldots, p_k\}$. Then for the LTS $L_{\mathcal{C}}$ we show there is a distinguishing trace smaller than $|S|$ for $s \in S$ and $t \in S$ if and only if $\mathcal{C}$ is satisfiable.

We begin by observing the sets $Tr(s)$, $Tr(t)$:

$$Tr(s) = \{\varepsilon, \mathtt{init}\} \cup \{\mathtt{init} \cdot w \mid w \in Tr(\bot_0) \cup Tr(\mathtt{sat}_0)\},$$
$$Tr(t) = \{\varepsilon, \mathtt{init}\} \cup \{\mathtt{init} \cdot w \mid w \in Tr(\mathtt{sat}_0) \cup \bigcup_{i \in [1,n]} Tr(\mathtt{unsat}_0^{C_i})\}.$$

Since for every $C \in \{C_1, \ldots, C_n\}$, $Tr(\mathtt{unsat}_0^C) \subseteq Tr(\bot_0)$ and $Tr(\mathtt{sat}_0) \subseteq Tr(\mathtt{unsat}_0^C)$, we know that if there is a distinguishing trace it has to be $\mathtt{init} \cdot w \in Tr(s)$ for a $w \in Tr(\bot_0)$. By Lemma 21 this trace w exists iff $\mathcal{C}$ is satisfiable. Hence, the states s and t are in *TRACE-DIST* if and only if $\mathcal{C}$ is in *CNF-SAT*. The LTS $L_{\mathcal{C}}$ can be computed in polynomial time, as it has $(n+2)(k+1)+3$ states and $2k(n+2)+2n+4$ transitions. This concludes the proof that *TRACE-DIST* is NP-complete. ◀

In the reduction a distinguishing trace is also a minimal distinguishing formula. Which means we can generalise our NP-hardness result.

▶ **Corollary 23.** *Deciding MIN-DIST is NP-hard.*

Proof. We prove this by a similar reduction as in the proof of Theorem 22. The intuition is that, given a CNF formula $\mathcal{C} = C_1 \wedge \ldots \wedge C_n$ with propositions $\mathit{Prop} = \{p_1, \ldots, p_k\}$, in the LTS $L_{\mathcal{C}}$ a distinguishing formula $\phi \in \mathcal{F}$ such that $|\phi| = k+2$ necessarily is a trace-formula.

We reduce CNF-SAT to MIN-DIST. Let $\mathcal{C} = C_1 \wedge \ldots \wedge C_n$ be a CNF formula over the propositions $\mathit{Prop} = \{p_1, \ldots, p_k\}$. Then for the LTS $L_{\mathcal{C}}$ we show there is a distinguishing formula $\phi \in \mathcal{F}$ for $s \in S$ and $t \in S$ such that $|\phi| \leqslant k+2$ if and only if $\mathcal{C}$ is satisfiable.

For the direction $\Rightarrow$, assume a formula $\phi \in \mathcal{F}$ exists such that $|\phi| \leqslant k+2$ and $s \not\sim_\phi t$. We show that this means $\mathcal{C}$ is satisfiable. We observe by the deterministic behaviour that $s \,\underline{\leftrightarrow}_{k+1}\, t$. Hence, by Theorem 10 we know $d_\diamond(\phi) \geq k+2$. Since we assume $|\phi| \leqslant k+2$ we know that $d_\diamond(\phi) = k+2$ and so, there are no non-trivial conjunctions, and we see that we can rewrite $\neg\neg\phi \mapsto \phi$. Hence, there is a formula $\psi = \triangle_1 \ldots \triangle_{k+2} tt$ such that for each $i \in [1, k+2]$, $\triangle_i \in \{\langle a_i \rangle, \neg\langle a_i \rangle\}$, for some $a_1, \ldots, a_{k+2} \in \mathit{Act}$, such that $[\![\phi]\!] = [\![\psi]\!]$.

By Lemma 17 there is a trace $w \in \mathit{Tr}(\psi)$, such that $|w| \geq k+2$ and, $w \in \mathit{Tr}(s) \cup \mathit{Tr}(t)$. The only trace of this length of s or t is in the shape $w = \texttt{init} \cdot \hat{p}_1 \ldots \hat{p}_k \cdot \texttt{false}$, where $\hat{p_i} \in \{p_i, \overline{p}_i\}$ for each $i \in [1, k]$. This means that $a_1 = \texttt{init}$, $a_{j+1} = \hat{p}_j$ for each $j \in [1, k]$ and $a_{k+2} = \texttt{false}$. We are going to show that the associated truth value $\rho = \rho_{\hat{p}_1 \ldots \hat{p}_k}$ satisfies $\mathcal{C}$ by reductio ad absurdum.

If ρ does not satisfy $\mathcal{C}$ then there is a clause C such that C is not satisfied by ρ. We claim for this clause $\texttt{unsat}_0^C \sim_{\Delta_2 \ldots \Delta_{k+1} tt} \bot_0$, and since both s and t have a $\texttt{init}$-transition to $\texttt{sat}_0$ this means ψ does not distinguish any of the derivatives. Hence $s \sim_\psi t$ which is a contradiction.

For the other direction if $\mathcal{C}$ is satisfiable then by Lemma 21 there is a $w \in \mathit{Act}^{k+1}$ such that $w \in \mathit{Tr}(\bot_0)$ and $w \notin \mathit{Tr}(\texttt{unsat}_0^C)$ for all clauses $C \in \{C_1, \ldots, C_n\}$. Using w we construct the distinguishing trace $w' = \texttt{init} \cdot w$. Since $w \in \mathit{Tr}(\bot_0)$, $w \notin \mathit{Tr}(\texttt{unsat}_0^C)$ and by construction also $w \notin \mathit{Tr}(\texttt{sat}_0)$, it is the case that $w' \in \mathit{Tr}(s)$ and $w' \notin \mathit{Tr}(t)$. This means the formula $\phi_{w'}$ is a distinguishing formula and $|\phi_{w'}| = k+2$, which finishes the second part of the proof. ◀

The problem MIN-DIST is not a member of NP since a polynomially sized witness might not exist. However, there is always a "shared" distinguishing formula of polynomial size. Since we can compute in polynomial time if a shared formula is a distinguishing formula, the decision problem MIN-DIST formulated in terms of total "shared" modalities is NP-complete.

4 Efficient algorithms

In this section we explain that despite the NP-hardness results from the previous section it is still possible to efficiently generate distinguishing formulas with minimal observation- and negation-depth. First, we introduce the method $\phi(s,t)$ listed in Algorithm 1 that generates a minimal observation-depth distinguishing formula for the states s and t. We extend $\phi(s,t)$ to the function $\psi_i(s,t)$ listed in Algorithm 2. This method computes a distinguishing formula with observation-depth of at most i and minimal negation-depth. Additionally, this procedure also prevents unnecessary conjuncts to be added. Finally, we indicate how to compute the equivalences $\underline{\leftrightarrow}_1, \ldots, \underline{\leftrightarrow}_k$, and the minimal observation- and negation-depth.

4.1 The algorithm

For every $i \in \mathbb{N}$, we define a function $\delta_i : S \times S \to 2^{\mathit{Act} \times S}$ that gives all distinguishing observations. More precisely, given two i-distinguishable states $s \in S$ and $t \in S$, $\delta_i(s,t)$ returns all pairs (a, s'), where $a \in \mathit{Act}$, $s' \in S$, such that $s \xrightarrow{a} s'$ and s' is $(i-1)$-distinguishable from all targets $t \xrightarrow{a} t'$. The definition of $\delta_i(s,t)$ is:

$$\delta_i(s,t) = \{(a, s') \mid s \xrightarrow{a} s' \text{ and } \forall t \xrightarrow{a} t'.\ \Delta(s', t') \leqslant i-1\}.$$

Algorithm 1 Minimal-depth distinguishing formula.

```
input  : Two states $s, t \in S$ such that $s \not\mathrel{\underline{\leftrightarrow}}_i t$
output : A formula $\phi \in \mathcal{F}$ s.t. $s \in [\![\phi]\!]$ and $t \notin [\![\phi]\!]$
Function $\phi(s,t)$ is
    $i := \Delta(s,t)$;
    if $\delta_i(s,t) = \emptyset$ then
        return $\neg\,\phi(t,s)$
    Select $(a,s') \in \delta_i(s,t)$;
    $T := \{t' \mid t \xrightarrow{a} t'\}$;
    return $\langle a \rangle \big( \bigwedge_{t' \in T} \phi(s',t') \big)$;
end
```

Using the function $\delta_i(s,t)$, we can compute a minimal observation-depth formula using the procedure listed as Algorithm 1. The procedure selects an action state pair $(a,s') \in \delta_i(s,t)$ and recursively distinguishes s' from all a-derivatives of t. If $\delta_i(s,t)$ is empty the negated $\phi_i(t,s)$ is calculated and in this case $\delta_i(t,s)$ is necessarily not empty.

▶ **Lemma 24.** *Given an LTS $L = (S, Act, \rightarrow)$ and two states $s, t \in S$. If $s \not\mathrel{\underline{\leftrightarrow}}_i t$ then: $\delta_i(s,t) \neq \emptyset$ or $\delta_i(t,s) \neq \emptyset$.*

Proof. As $s \not\mathrel{\underline{\leftrightarrow}}_i t$ there either is an $s \xrightarrow{a} s'$ such that $s' \not\mathrel{\underline{\leftrightarrow}}_{i-1} t'$ for all $t \xrightarrow{a} t'$, or vice-versa there is a $t \xrightarrow{a} t'$ such that $t' \not\mathrel{\underline{\leftrightarrow}}_{i-1} s'$ for all $s \xrightarrow{a} s'$. In the first case $(a,s') \in \delta_i(s,t)$, in the second case $(a,t') \in \delta_i(t,s)$. ◀

4.2 Minimal negation-depth

In order to minimize the number of negations within the minimal observation-depth formula we combine the notions of k-bisimilar and m-nested similarity inclusion.

▶ **Definition 25.** *Let $L = (S, Act \rightarrow)$ be an LTS, and $k, m \in \mathbb{N}$. We define m-nested k-similarity inclusion, denoted $\mathrel{\underline{\rightarrow}}^m_k$, inductively by for all $s, t \in S$, $s \mathrel{\underline{\rightarrow}}^m_0 t$ and if $s \mathrel{\underline{\rightarrow}}^m_k t$ then*

1. *if $s \xrightarrow{a} s'$ there is a $t \xrightarrow{a} t'$ such that $s' \mathrel{\underline{\rightarrow}}^m_{k-1} t'$, and*
2. *if $m > 0$ and $t \xrightarrow{a} t'$, then there is a $s \xrightarrow{a} s'$ such that $t' \mathrel{\underline{\rightarrow}}^{m-1}_{k-1} s'$.*

Similarly to the original Hennessy-Milner correspondences, we observe the correspondence between the fragment $\mathcal{F}^m_k$ and the relation $\mathrel{\underline{\rightarrow}}^m_k$.

▶ **Theorem 26.** *Let $L = (S, Act, \rightarrow)$ be an LTS. For any $k, m \in \mathbb{N}$ and states $s, t \in S$:*

$$s \leqslant_{\mathcal{F}^m_k} t \iff s \mathrel{\underline{\rightarrow}}^m_k t.$$

Related to the distance measure Δ, we define the directed minimal negation-depth measure for the relation $\mathrel{\underline{\rightarrow}}^m_k$, for states that are not m-nested k-similar for some $k, m \in \mathbb{N}$.

▶ **Definition 27.** *Let $L = (S, Act, \rightarrow)$ be an LTS and $i \in \mathbb{N}$ be a number. We define the directed minimal negation-depth $\overrightarrow{\Delta}_i : S \times S \rightarrow \mathbb{N} \cup \{\infty\}$ by*

$$\overrightarrow{\Delta}_i(s,t) = \begin{cases} j & \text{if } s \not\mathrel{\underline{\rightarrow}}^j_i t, \text{ and } s \mathrel{\underline{\rightarrow}}^{j-1}_i t, \\ \infty & \text{if } s \mathrel{\underline{\leftrightarrow}}_i t. \end{cases}$$

Algorithm 2 Generate a distinguishing formula with minimal observation- and negation-depth.

```
   input : Two states $s, t \in S$ such that $s \not\equiv_i t$ for some $i \in \mathbb{N}$
   output: A formula $\phi \in \mathcal{F}_i$ such that $s \in [\![\phi]\!]$ and $t \notin [\![\phi]\!]$
1  Function $\phi_i(s,t)$ is
2      $j := \overrightarrow{\Delta}_i(s,t)$;
3      $\mathcal{X} := \hat{\delta}_i^j(s,t)$;
4      if $\mathcal{X} = \emptyset$ then
5          return $\neg\,\phi_i(t,s)$
6      Select $(a, s') \in \mathcal{X}$;
7      $T := \{t' \mid t \xrightarrow{a} t'\}$;
8      while $T \neq \emptyset$ do
9          Select $t_{max} \in T$ such that $\overrightarrow{\Delta}_{i-1}(s', t_{max}) \geq \overrightarrow{\Delta}_{i-1}(s', t')$ for all $t' \in T$;
10         $\phi_{t_{max}} := \phi_{i-1}(s', t_{max})$;
11         $\Phi := \Phi \cup \{\phi_{t_{max}}\}$;
12         $T := T \cap [\![\phi_{t_{max}}]\!]$;
13     end
14     return $\langle a \rangle \left( \bigwedge_{\phi \in \Phi} \phi \right)$
15 end
```

For every $i, j \in \mathbb{N}$ we define a function $\hat{\delta}_i^j : S \times S \to 2^{Act \times S}$ that is similar to the function δ_i. It adds an extra limitation on the number of negations needed to distinguish the pairs from all observations from t.

$$\hat{\delta}_i^j(s,t) = \{(a, s') \mid (a, s') \in \delta_i(s,t) \text{ and } \forall t \xrightarrow{a} t'.\ \overrightarrow{\Delta}_{i-1}(s', t') \leqslant j\}.$$

The next lemma guarantees that a suitable distinguishing observation exists.

▶ **Lemma 28.** *Given an LTS $L = (S, Act, \to)$ and two states $s, t \in S$. Then for all $i, j \in \mathbb{N}$, if $s \not\equiv_i^j t$ then $\hat{\delta}_i^j(s,t) \neq \emptyset$ or $\hat{\delta}_i^{j-1}(t,s) \neq \emptyset$.*

In Algorithm 2 we give the method $\psi_i(s,t)$ that given an LTS $L = (S, Act, \to)$ and i-distinguishable states $s, t \in S$ generates a formula such that $s \in [\![\psi_i(s,t)]\!]$ and $t \notin [\![\psi_i(s,t)]\!]$ with observation depth at most i and minimal negation-depth.

The algorithm attempts to find an action label $a \in Act$ and an a-derivative $s \xrightarrow{a} s'$, such that all a-derivatives t', such that $t \xrightarrow{a} t'$ are distinguishable with a formula with at most $i-1$ nested observations and j nested negations. These pairs (a, s') are given by the function $\hat{\delta}_i^j(s,t)$. In Line 6 one of these witnesses is chosen. If there is more than one suitable derivate, one is chosen at random.

The next theorem states that Algorithm 2 yields a valid distinguishing formula.

▶ **Theorem 29.** *Let $L = (S, Act, \to)$ be an LTS, and $s, t \in S$ be states. If s and t are k-distinguishable for some $k \in \mathbb{N}$ then $s \in [\![\psi_k(s,t)]\!]$ and $t \notin [\![\psi_k(s,t)]\!]$.*

The next theorem states that if $\Delta(s,t) = k$, then $\psi_k(s,t)$ yields a formula that has minimal observation-depth, and there is no formula ϕ with a smaller number of nested negations such that $s \not\leqslant_\phi t$.

▶ **Theorem 30.** *Let $L = (S, Act, \to)$ be an LTS, and $s, t \in S$ be states, such that $s \not\equiv t$ and $\Delta(s,t) = k$. Then for all $\phi \in \mathcal{F}$, if $s \not\leqslant_\phi t$ then $d_\diamond(\psi_k(s,t)) \leqslant d_\diamond(\phi)$ and if $d_\neg(\phi) < d_\neg(\psi_k(s,t))$ then $d_\diamond(\phi) > d_\diamond(\psi_k(s,t))$.*

Algorithm 3 Iterative partition refinement.

```
Function Refine($\pi$) is
    $\pi' := \pi$;
    foreach $a \in Act, B' \in \pi$ do
        foreach $B \in \pi'$ do
            $C := split_a(B, B')$;
            if $C \neq B$ and $C \neq \emptyset$ then
                $\pi' := (\pi' \setminus \{B\}) \cup \{C, B \setminus C\}$;
    return $\pi'$;
$i := 0$; $\pi_0 := \{S\}$;
while $\pi_i \neq$ Refine($\pi_i$) do
    $\pi_{i+1} :=$ Refine($\pi_i$);
    $i := i + 1$;
```

4.3 Partition refinement

In order to execute Algorithm 2, we need to compute the functions Δ and $\overrightarrow{\Delta}$. In this section we propose a simple partition refinement algorithm that does exactly this by first computing the relations $\underline{\leftrightarrow}_0, \underline{\leftrightarrow}_1, \ldots, \underline{\leftrightarrow}_k$ iteratively. The pseudocode is listed in Algorithm 3. In contrast to the more efficient partition refinement algorithms [12, 21, 24], we guarantee that *older* blocks are used first as splitter. This method is inspired by [23] where pairwise minimal distinguishing words are computed.

Most algorithms deciding bisimilarity are so-called partition refinement algorithms [14, 21]. Our algorithms are also based on partition refinement. A *partition* π of a set S is a disjoint cover of S, i.e. a set of non-empty subsets of S and every element of S is in exactly one subset. The elements $B \in \pi$ are called *blocks*. A partition π induces the equivalence relation $\sim_\pi \colon S \times S$ in which the blocks are the equivalence classes, i.e. $\sim_\pi = \{(s,t) \mid \exists B \in \pi \text{ and } s,t \in B\}$. In the algorithm we filter a set of states U on a distinguishing observation with respect to a set of given states V, and an action $a \in Act$, i.e.: $split_a(U,V) = \{s \in U \mid \exists s' \in V. s \xrightarrow{a} s'\}$.

The next theorem states that the procedure listed as Algorithm 3 produces a sequence of partitions, in which the i-th partition induces i-bisimilarity.

▶ **Theorem 31.** *Given an LTS $L = (S, Act, \rightarrow)$ and partitions $\pi_0, \ldots, \pi_k$ produced by Algorithm 3. Then $\sim_{\pi_i} = \underline{\leftrightarrow}_i$, for all $0 \leqslant i \leqslant k$.*

It is possible to compute the function $\overrightarrow{\Delta}_i(s,t)$ in polynomial time from the computed k-bisimilarity relations calculated in Algorithm 3. It is important to use dynamic programming such that $\overrightarrow{\Delta}_i(s,t)$ for every i, s and t is only calculated once.

4.4 Evaluation

The computation of Algorithm 2 needs to account for redundancies to guarantee a polynomial time algorithm. We use dynamic programming to achieve this. For any pair of states $s,t \in S$ if the function $\psi_i(s,t)$ is invoked, it stores the generated shared formula. Whenever the function is called again, the previously generated formula is used, with only constant extra computing and memory usage. Hence, given an LTS $L = (S, Act, \rightarrow)$ the number of recursive calls is limited to the combination of states and level $k \leqslant |S|$, i.e. $\mathcal{O}(|S|^3)$ calls.

▶ **Corollary 32.** *Given an LTS $L = (S, Act, \rightarrow)$ and a pair of distinguishable states $s,t \in S$, then the following is computable in polynomial time:*

- *A minimal observation-depth distinguishing formula,*
- *A minimal observation- and negation-depth distinguishing formula.*

A naive implementation of the algorithms requires quadratic memory. This could be a bottleneck for large state spaces. Representing the equivalences $\underline{\leftrightarrow}_k$ as a splitting tree [17] is more memory efficient. In addition, an optimization is to generate only distinguishing formulas between equivalence classes of the generated equivalences, instead of individual states.

Table 1 Results from prototype implementation Algorithm 1.

	Max						Average					
Benchmark	$d_\Diamond(\phi)$		$\lvert\phi\rvert$		$d_\neg(\phi)$		$d_\Diamond(\phi)$		$\lvert\phi\rvert$		$d_\neg(\phi)$	
	Our	Cleav.	Our	Cleav.	Our	Cleav.	Our	Cleav.	Our	Cleav.	Our	Cleav.
ieee-1394-1	64	891	69	1355	0	886	64,0	247,2	69,0	373,7	0,0	243,2
ieee-1394-2	37	224	42	320	1	219	37,0	92,0	42,0	120,0	1,0	88,2
ieee-1394-3	102	698	102	1092	2	696	102,0	299,1	102,0	465,4	2,0	295,7
ieee-1394-4	76	363	83	506	2	360	76,0	196,6	80,9	276,5	2,0	194,5
ieee-1394-5	18	155	18	214	2	146	18,0	36,0	18,0	44,8	2,0	30,4

We implemented a prototype of the method introduced here. We also implemented the method proposed by Cleaveland [6] in which we decided bisimilarity by a partition refinement algorithm in which the splitter selected is the latest created block, since heuristically this has the best runtime [1, 2]. For Cleaveland's method the strategy for splitter selection matters for the size of the formulas generated. However, regardless of strategy chosen, the formulas that our method generates are always more concise in all metrics.

We post-processed the formulas to ensure both implementations resulted in formulas that are irreducible. For the benchmark we used the model from [18] containing 188.568 states and 340.607 transitions. We compared this model to 5 modified versions where we omitted one randomly chosen transition. In Table 1 the results of running the algorithms 10 times are shown. Under "Max", the worse-case of the different runs for each metric is listed for our method ("Our"), next to the result of the implementation of Cleaveland ("Cleav."). Under "Average" the average of the 10 runs is shown.

We see that our new method consistently outputs a minimal observation- and negation-depth formula, and the generated formulas only rarely deviates in size. It outperforms the method of Cleaveland in all cases. In some cases the depth is improved a factor 10.

5 Conclusions & Future work

In this work we studied the problem of computing minimal distinguishing formulas. We introduced three metrics: size, observation-depth, and negation-depth. Using a reduction directly from CNF-SAT we showed that finding a minimal sized distinguishing formula is NP-hard. However, for observation- and negation-depth, we introduce polynomial time algorithms that compute minimal formulas. A prototype demonstrates the potential improvement over the method introduced by Cleaveland [6]. A more rigorous version is implemented in the mCRL2 toolset [5].

For future work it would be interesting to extend our algorithms for equivalences beyond strong bisimilarity. For instance, a more generic coalgebraic treatment, extending [25], or computing smaller witnesses for equivalences with abstractions like branching and weak bisimilarity, improving upon the work of Korver [16].

References

1 Manuel Baclet and Claire Pagetti. Around Hopcroft's algorithm. In *Implementation and Application of Automata: 11th International Conference, CIAA 2006, Taipei, Taiwan, August 21-23, 2006. Proceedings 11*, pages 114–125. Springer, 2006.

2 Christoph Berkholz, Paul Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory of Computing Systems*, 60(4):581–614, 2017.

3 Benjamin Bisping, David N. Jansen, and Uwe Nestmann. Deciding all behavioral equivalences at once: A game for linear-time–branching-time spectroscopy. *Logical Methods in Computer Science*, 18(3), August 2022. doi:10.46298/lmcs-18(3:19)2022.

4 Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1):115–131, 1988. doi:10.1016/0304-3975(88)90098-9.

5 Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '19)*, pages 21–39. Springer International Publishing, 2019. doi:10.1007/978-3-030-17465-1_2.

6 Rance Cleaveland. On automatically explaining bisimulation inequivalence. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proc. Computer-Aided Verification (CAV 1990)*, pages 364–372, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. doi:10.1007/BFb0023750.

7 Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-Wei Wang. Regular expressions: New results and open problems. *J. Autom. Lang. Comb.*, 10(4):407–437, 2005.

8 Santiago Figueira and Daniel Gorín. On the size of shortest modal descriptions. In *Advances in Modal Logic*, volume 8, pages 114–132, 2010.

9 Herman Geuvers. Apartness and distinguishing formulas in Hennessy-Milner logic. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning*, pages 266–282. Springer Nature Switzerland, 2022. doi:10.1007/978-3-031-15629-8_14.

10 Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992. doi:10.1016/0890-5401(92)90013-6.

11 Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming (ICALP '1980)*, pages 299–309, Berlin, Heidelberg, 1980. Springer-Verlag. doi:10.5555/646234.758793.

12 John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971. doi:10.1016/B978-0-12-417750-5.50022-1.

13 Harry B. Hunt III. *On the Time and Tape Complexity of Languages*. PhD thesis, Cornell University, 1973.

14 Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990. doi:10.1016/0890-5401(90)90025-D.

15 Barbara König, Christina Mika-Michalski, and Lutz Schröder. Explaining non-bisimilarity in a coalgebraic approach: Games and distinguishing formulas. In Daniela Petrişan and Jurriaan Rot, editors, *Coalgebraic Methods in Computer Science (CMCS 2022)*, pages 133–154. Springer, 2020.

16 Henri Korver. Computing distinguishing formulas for branching bisimulation. In Kim G. Larsen and Arne Skou, editors, *Computer Aided Verification (CAV 1991)*, pages 13–23. Springer, 1992.

17 David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on computers*, 43(3):306–320, 1994. doi:10.1109/12.272431.

18 Bas Luttik. *Description and formal specification of the Link Layer of P1394*. CWI report. SEN-R : software engineering. Centrum voor Wiskunde en Informatica, 1997.

19 Robin Milner. *A calculus of communicating systems*. Springer, 1980.

20 Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

21 Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. doi:10.1137/0216062.

22 David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, pages 167–183. Springer Berlin Heidelberg, 1981.

23 Rick Smetsers, Joshua Moerman, and David N Jansen. Minimal separating sequences for all pairs of states. In Adrian-Horia Dediu, Jan Janoušek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications (LATA 2016)*, pages 181–193. Springer, 2016.

24 Antti Valmari. Bisimilarity minimization in o(m logn) time. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and Theory of Petri Nets*, pages 123–142, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

25 Thorsten Wißmann, Stefan Milius, and Lutz Schröder. Quasilinear-time computation of generic modal witnesses for behavioural inequivalence. *Logical Methods in Computer Science*, 18(4), November 2022. doi:10.46298/lmcs-18(4:6)2022.

26 Jacob K. Wortmann, Simon R. Olesen, and Søren Enevoldsen. Caal 2.0. recursive HML, distinguishing formulae, equivalence collapses and parallel fixed-point computations. Master's thesis, Aalborg University, 2015.

Complexity of Membership and Non-Emptiness Problems in Unbounded Memory Automata

Clément Bertrand
Scalian Digital Systems, Valbonne, France

Cinzia Di Giusto[1]
Université Côte d'Azur, CNRS, I3S, France

Hanna Klaudel
IBISC, Univ. Evry, Université Paris-Saclay, France

Damien Regnault
IBISC, Univ. Evry, Université Paris-Saclay, France

Abstract

We study the complexity relationship between three models of unbounded memory automata: *nu*-automata (ν-A), Layered Memory Automata (LaMA)and History-Register Automata (HRA). These are all extensions of finite state automata with unbounded memory over infinite alphabets. We prove that the membership problem is NP-complete for all of them, while they fall into different classes for what concerns non-emptiness. The problem of non-emptiness is known to be Ackermann-complete for HRA, we prove that it is PSPACE-complete for ν-A.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Theory of computation → Problems, reductions and completeness

Keywords and phrases memory automata, ν-automata, LaMA, HRA, complexity, non-emptiness, membership

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.33

Related Version *Full Version*: `https://hal.science/hal-04155339/file/main.pdf` [3]

Acknowledgements We want to thank the anonymous reviewers for the careful reviews and insightful comments.

1 Introduction

We study unbounded memory automata for words over an infinite alphabet, as introduced in [13, 17]. Such automata model essentially dynamic generative behaviours, i.e., programs that generate an unbounded number of distinct resources each with its own unique identifier (e.g. thread creation in Java, XML). For a detailed survey, we refer the reader to [4, 15]. We focus, in particular, on three formalisms, ν-automata (ν-A) [9, 5, 8], Layered Memory Automata (LaMA) [4] and HRA for History-Register Automata [11]. All these models are extensions of finite state automata with memory-storing letters. The memory for HRA is composed of registers (that can store only one letter) and histories (that can store an unbounded number of letters). Whereas the memory for the other two models consists of a finite set of variables. Among the distinctive features of HRA, they can reset registers and histories, and select, remove and transfer individual letters. In ν-A and LaMA, variables must satisfy an additional constraint, referred to as injectivity, meaning that they cannot store shared letters. Moreover, variables can be emptied (reset) but single letters cannot be removed. We know that LaMA are more expressive than ν-A as the former are closed under intersection while it is not the case for the latter ones.

[1] Corresponding author

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 33; pp. 33:1–33:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

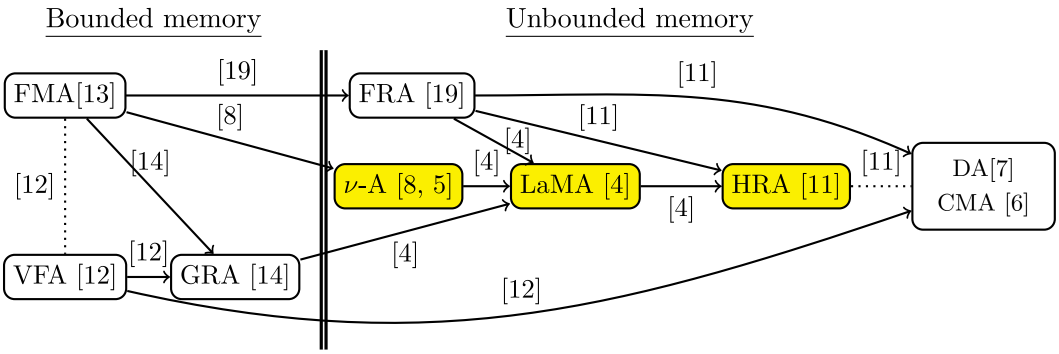


Figure 1 A classification of memory automata from [4]. Arrows represent strict language inclusion, while dotted lines denote language incomparability. The formalisms studied here are in yellow.

We tackle two problems: membership and non-emptiness. From a practical point of view, automata over infinite alphabets can be used to identify patterns in link-stream analysis [5]. In such a scenario, the alphabet is not known in advance (open systems) and runtime verification can help to recognize possible attacks on a network by looking for specific patterns. This problem corresponds to checking whether a pattern (word) belongs to a language (the membership problem). Concerning non-emptiness, this is the "standard" problem to address while considering automata in general.

Fig. 1 depicts the unbounded memory automata known in the literature (to the best of our knowledge). An implementation exists for ν-A and LaMA which includes an implementation of the membership algorithm, but we have not found anything neither for Data Automata (DA) nor Class Memory Automata (CMA). Both DA and CMA are incomparable classes wrt to HRA, hence we chose not to consider them. Fresh-Register automata (FRA), ν-A, LaMA and HRA are instead related from the expressiveness point of view. Given the similarities between those formalisms, the existence of implementations and the lack of complexity results we find it natural to consider these classes of automata.

Application-wise, ν-A, called resource graphs in [9], model the use of unbounded resources in the π-calculus, aiming at minimizing them. Then, runtime verification on link-streams was the initial motivation for the introduction of (timed)ν-A [5]. In subsequent works, LaMA have been introduced to be able to construct the synchronous product, hence being able to express the synchronization of resources. This entails the closure by intersection, which is interesting when one wants to define a language of expressions, an extension of (timed) regular expressions, which was proposed in the PhD thesis of Clément Bertrand [2].

For a precise discussion on the relations among these formalisms see [4], here we just recall the hierarchy: cfr. Fig. 1. Apart from expressiveness, a number of questions concerning complexity remains open. We know that checking whether the language recognized by an HRA is empty or not (referred to as the non-emptiness problem) is Ackermann-complete [11]. But the question has not been addressed for ν-A and LaMA. For finite-memory automata (FMA), it is known that membership (testing whether a word belongs to the language) and non-emptiness are NP-complete [18]. Knowing whether a language is included in another is undecidable for FMA when considering their non-deterministic version, but it is PSPACE-complete for deterministic ones [16]. In [12] it is shown that the non-emptiness problem for Variable Finite Automata (VFA) is NL-complete, while membership is NP-complete. For FRA and Guessing-Register Automata (as they are called in [15]) we only know that both problems are decidable but we do not know the accurate complexity class. Finally, for data-languages where data-words are sequences of pairs of a letter from a finite alphabet and an element from an infinite set and the latter can only be compared for equality, the non-emptiness problem for FMA is PSPACE-complete [10], for DA and CMA, membership and non-emptiness are only shown to be decidable, but no complexity is given [7, 6].

Contributions. In this paper, we close some open problems on the complexity of membership and non-emptiness. We first prove that testing membership for HRA, ν-A and LaMA is an equivalent problem. Then we address complexity and show that the problem is NP-complete with a reduction of 3-SAT to LaMA. Non-emptiness appears to be a much harder problem. We show that the non-emptiness problem is PSPACE-complete for ν-A by reducing TQBF (True Fully Quantified Boolean Formula) to ν-A.

The paper is organized as follows. The three formalisms are introduced and the main differences are recalled quickly in Section 2. Section 3 presents the complexity of the membership problem and Section 4 the non-emptiness one. Finally, Section 5 concludes with some remarks. Proofs and additional material can be found in [3].

2 Formalisms

All three formalisms are generalizations of finite state automata with memory over an infinite alphabet $\mathcal{U}$. For all of them, configurations (q, M) are pairs of a state of the automaton plus a memory context. A memory context assigns a set of letters to each identifier of the memory, variable or history depending on the formalism under consideration.

▶ **Definition 1** (Memory context). *Given a finite set of memory identifiers or variables V and an infinite alphabet $\mathcal{U}$, we define a memory context M as an assignment: $M : V \to 2^{\mathcal{U}}$ where $M(v) \subset \mathcal{U}$ is the finite set of letters* assigned *to v.*

The definition of accepted language common to the three formalisms is, as customary:

▶ **Definition 2** (Accepted language). *For an automaton A (LaMA, ν-A or HRA), the language of A is the set of words recognized by A: $\mathcal{L}(A) = \{w \in \mathcal{U}^* \mid (q_0, M_0) \xRightarrow[A]{w} (q_f, M)$ s.t. $q_f \in F\}$, where $\xRightarrow[A]{w}$ is the extension to sequences of transitions of $\xrightarrow[A]{u}$.*

2.1 n-Layered Memory Automata

We start with n-Layered Memory Automata (n-LaMA). The idea is that finite state automata are enriched with n layers each containing a finite number of variables. Variables on the same layer satisfy the *injectivity* constraint: variables on a given layer $l \in [1, n]$ (denoted v^l) do not share letters of the alphabet: $\forall v_1 \neq v_2 \in V, M(v_1^l) \cap M(v_2^l) = \emptyset$. Upon reading a letter, a transition can test if the letter is already stored in a set of variables and add a letter to a set of variables. The non-observable transition (ε-transition) empties a set of variables.

▶ **Definition 3** (n-LaMA). *An n-LaMA A is defined by the tuple $(Q, q_0, F, \Delta, V, n, M_0)$, where:*

- *Q is a finite set of states,*
- *$q_0 \in Q$ and $F \subseteq Q$ are respectively an initial state and a set of final states,*
- *Δ is a finite set of transitions,*
- *n is the number of layers, and V is a finite set of variables, denoted v^l with $l \in [1, n]$,*
- *$M_0 : V \mapsto 2^{\mathcal{U}}$ is an initial memory context.*

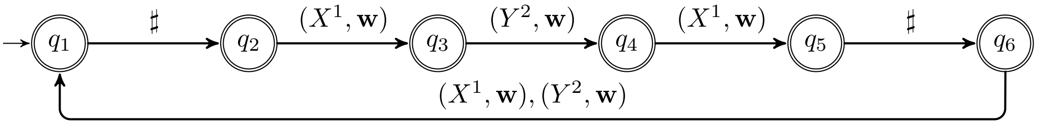


Figure 2 A 2-LaMA A_p recognizing $\mathcal{P}(2) \cap \mathcal{P}(3)$ from Example 8.

A *fresh letter at layer l*, is a letter that is associated with no variable of this layer. The set of transitions $\Delta = \Delta_o \cup \Delta_\varepsilon$ encompasses two kinds of transitions with Δ_o the set of *observable transitions* that consume a letter of the input and Δ_ε the set of *non-observable transitions*, that do not consume any letter of the input but can reset a set of variables.

▶ **Definition 4** (Observable transition). *An observable transition is a tuple of the form: $\delta = (q, \alpha, q') \in \Delta_o$ where:*

- *$q, q' \in Q$ are the source and destination states of the transition,*
- *$\alpha : [1, n] \to (V \times \{\mathbf{r}, \mathbf{w}\}) \cup \{\sharp\}$, such that $\alpha(l) = (v^l, \mathbf{x})$ for $\mathbf{x} \in \{\mathbf{r}, \mathbf{w}\}$ and for some $v^l \in V$ indicates for each layer l which variable is examined by the transition.*

Notice that only one variable per layer can be examined, and it is not possible to have $\alpha(l) = (v^k, \mathbf{x})$ with $l \neq k$. The special symbol $\sharp$ indicates that no variable is to be read or written for a specific layer.

▶ Remark 5 (Any-letter transition). If $\forall l \in [1, n], \alpha(l) = \sharp$ (i.e., no variable is examined) then the transition is executed consuming whatever letter is in input.

▶ **Definition 6** (Non-observable transition). *A non-observable transition is a tuple of the form $\delta_\varepsilon = (q, \mathbf{reset}, q') \in \Delta_\varepsilon$ where:*

- *$q, q' \in Q$ are the source and destination states of δ_ε,*
- *$\mathbf{reset} \subseteq 2^V$ is the set of variables reset (i.e., emptied) by the transition.*

▶ **Definition 7.** *The semantics of an n-LaMA $A = (Q, q_0, F, \Delta, V, n, M_0)$ is defined as:*

- *An observable transition (q, α, q') can be executed on an input letter u from memory context M leading to M': $(q, M) \xrightarrow[A]{u} (q', M')$ if for each $\alpha(l) \neq \sharp$ such that $\alpha(l) = (v^l, \mathbf{x})$:*
  - *if $\mathbf{x} = \mathbf{r}$, then $u \in M(v^l)$ and $M'(v^l) = M(v^l)$;*
  - *if $\mathbf{x} = \mathbf{w}$, then u is fresh for layer l and u is added to v^l in the reached memory context: $M'(v^l) = M(v^l) \cup \{u\}$.*

  All variables v^l not labeled through α remains associated to the same letters : if $\alpha(l) = \sharp$ or $\alpha(l) = (v^l_1, x)$ and $v^l_1 \neq v^l$ then $M'(v^l) = M(v^l)$.
- *A non-observable transition $(q, \mathbf{reset}, q')$ can be executed from memory context M without reading any input letter leading to M': $(q, M) \xrightarrow[A]{\varepsilon} (q', M')$, where $\forall v^l \in \mathbf{reset} : M'(v^l) = \emptyset$ and otherwise $M'(v^l) = M(v^l)$.*

▶ **Example 8.** Let $\mathcal{P}(p) = \{u_1 \dots u_s \mid \forall j, k > 0, j \neq k,\ u_{j \cdot p} \neq u_{k \cdot p}\}$, be the language recognizing words where the letters at positions, which are multiples of p are all different whereas the others are not constrained. Fig. 2 depicts a 2-LaMA for $\mathcal{P}(2) \cap \mathcal{P}(3)$.

2.2 ν-automata

ν-automata (ν-A) can be seen as a restricted version of LaMA with only one layer. Hence, each variable is constrained under the injectivity property, and no letter can be stored in more than one variable.

▶ **Definition 9** (ν-A)**.** *A ν-A is defined as a tuple $(Q, q_0, F, \Delta, V, M_0)$, where*

- *Q is a finite set of states containing an initial state $q_0 \in Q$ and a set of final ones $F \subseteq Q$,*
- *V is a finite set of variables that may initially be storing a finite amount of letters from the infinite alphabet $\mathcal{U}$, as specified by the initial memory context M_0,*
- *and Δ is a finite set of transitions.*

As before, $\Delta = \Delta_o \cup \Delta_\varepsilon$ where Δ_o is the set of *observable transitions* and Δ_ε is the set of *non-observable* ones. Differently from LaMA, observable transitions are decoupled in read and write transitions.

▶ **Definition 10** (Observable transition)**.** *An observable transition can be of two kinds: $(q, v, \mathbf{r}, q')$ and $(q, v, \mathbf{w}, q')$ ($\mathbf{r}$ for read and $\mathbf{w}$ for write) where $q, q' \in Q$ are the source and destination states and $v \in V$.*

▶ **Definition 11** (Non-observable transition)**.** *A non-observable transition is a tuple of the form $\delta_\varepsilon = (q, \mathbf{reset}, q') \in \Delta_\varepsilon$ where: $q, q' \in Q$ are the source and destination states of δ_ε, $\mathbf{reset} \subseteq 2^V$ is the set of variables reset by the transition.*

▶ **Definition 12.** *The semantics of a ν-A $A = (Q, q_0, F, \Delta, V, M_0)$ is defined as:*

- *An observable transition $(q, v, \mathbf{x}, q')$ reading input letter u can be executed from memory context M leading to M': $(q, M) \xrightarrow[A]{u} (q', M')$ if for each v:*
  - *if $\mathbf{x} = \mathbf{r}$, then $u \in M(v)$ and $M'(v) = M(v)$;*
  - *if $\mathbf{x} = \mathbf{w}$, then u is fresh in M and u is added to v in the reached memory context: $M'(v) = M(v) \cup \{u\}$.*

  All other variables $v_1 \neq v$, remains associated to the same letters $M'(v_1) = M(v_1)$.
- *A non-observable transition $(q, \mathbf{reset}, q') \in \Delta_\varepsilon$ can be executed from memory context M leading to M': $(q, M) \xrightarrow[A]{\varepsilon} (q', M')$ without reading any input letter, where $\forall v \in \mathbf{reset}$: $M'(v) = \emptyset$ and otherwise $M'(v) = M(v)$.*

▶ Remark 13. Analogously to LaMA, we consider *any-letter transitions*, denoted by $(q, \sharp, q')$ with $\sharp \notin \mathcal{U}$, which are enabled whenever a letter is read and the memory context of the target configuration is the same as the origin's one.

Notice that any-letter transitions do not alter the expressive power of ν-A nor the complexity of its problems. Indeed, it is a sort of macro that can be encoded by a set of transitions searching for the presence of a letter or its freshness over the whole set V. To do so, one needs as many reading transitions as variables to allow the firing with any letter in memory. For fresh letters, one needs a transition writing in an extra variable, which is reset immediately after.

2.3 History-Register Automata

HRA are automata provided with a finite set H of histories, i.e., variables storing a finite subset of letters of the infinite alphabet $\mathcal{U}$. To simplify the presentation, we consider HRA defined only with histories and no registers. The latter does not provide additional expressiveness [11]. An essential distinction between HRA and LaMA or ν-A is that different histories are allowed to store the same letter (i.e., there is no injectivity constraint). Thus, an observable transition is annotated with the exact set of histories that should contain the letter read to enable it. This entails that for each observable transition the whole memory has to be explored while LaMA allow ignoring some layers using symbol $\sharp$ (this can be crucial while implementing the formalisms[2]).

[2] Implemented in tool available at `https://github.com/clementber/MaTiNA/tree/master/LaMA`

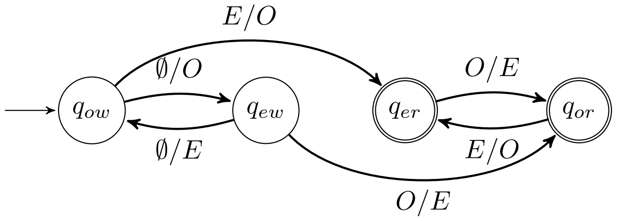


Figure 3 Example of an HRA A_h.

▶ **Definition 14** (HRA). *A History-Register Automata is defined as a tuple of the form $A = (Q, q_0, F, \Delta, H, M_0)$ where Q is the set of states, q_0 the initial one, F the set of final ones, Δ the set of transitions, H a finite set of histories and M_0 the initial memory context. The set of transitions $\Delta = \Delta_o \cup \Delta_\varepsilon$ are of the form:*

- *$(q, H_r, H_w, q') \in \Delta_o$ where $H_r, H_w \subseteq H$ (for read and write), which is an observable transition and*
- *$(q, H_\emptyset, q') \in \Delta_\varepsilon$ where $H_\emptyset \subseteq H$, which is a non-observable transition.*

An observable transition (q, H_r, H_w, q') is enabled if letter u is present in exactly all the histories in H_r and not present in $H \setminus H_r$. After the transition, u is present only in the histories in H_w. Notice that this allows moving an input letter from one set of histories to another, or even forgetting it if $H_w = \emptyset$. This is not possible in ν-A and LaMA. Finally, if $H_r = \emptyset$ then the input letter has to be fresh (absent from every history).

▶ **Definition 15.** *The semantics of an HRA $A = (Q, q_0, F, \Delta, H, M_0)$ is defined as:*

- *an observable transition (q, H_r, H_w, q') is enabled for memory context M when reading letter $u \in \mathcal{U}$: $(q, M) \xrightarrow[A]{u} (q', M')$ if $u \in M(h_r) \Leftrightarrow h_r \in H_r$ and $\forall h_w \in H_w : M'(h_w) = M(h) \cup \{u\}$ and $\forall h \notin H_w$: $M'(h) = M(h) \setminus \{u\}$;*
- *a non-observable transition $(q, H_\emptyset, q')$ is enabled for any memory context M and allows to move from configuration (q, M) to (q', M'): $(q, M) \xrightarrow[A]{\varepsilon} (q', M')$, where all the histories in $H_\emptyset$ have been reset in M'.*

▶ **Example 16.** Fig. 3 depicts an HRA that, with an initially empty memory context, recognizes the language

$$\begin{aligned}\{u_1u_2\dots u_n \mid\ & \exists k < n, \forall i, j \in [1, k], u_i = u_j \Leftrightarrow i = j,\\ & \forall m \in]k, n], \exists p < m, u_p = u_m, p \bmod 2 \neq m \bmod 2, \nexists q \in]p, m[, u_m = u_q\}\end{aligned}$$

The two transitions looping between states q_{ow} and q_{ew} allow us to recognize words where the first k letters are all different from each other. Letters are stored in histories O (odd) and E (even) to remember the parity of the position they are read at. The transitions between states q_{er} and q_{or} allow us to recognize words whose suffix is only composed of repetitions of the k first letters, with the additional constraint that they can only occur at a position with opposed parity wrt the previous occurrence. Thus, if a letter was read for the last time at an even position, it is stored in history E and can only be read in an odd position. Once it is read, it is transferred to the O history to remember it can only be read at an even position the next time.

3 Complexity of the membership problem

We know that each ν-A can be encoded into a LaMA and respectively each LaMA can be encoded into an HRA both recognizing the same language [4]. The encoding from LaMA to HRA is exponential in the number of layers, hence we know that the complexity of problems for HRA gives an upper bound to the complexity of the same problem for LaMA and ν-A. In this section, we show that the complexity of the membership problem (i.e., given an automaton A and a word w decide whether $w \in \mathcal{L}(A)$) falls in the same class for these three automata models. To do so, we show that the membership problem for LaMA can be simulated using ν-A, and the same can be done for HRA using LaMA.

Simulating the membership for LaMA in ν-automata. The idea is to represent an n-LaMA as a product of n ν-A, one for each layer. The main limitation is that having just one layer makes the injectivity constraint stronger. Indeed, it is not possible to trivially treat a same letter stored on different layers. To cope with this difficulty, we rename the word under consideration, replacing consistently each letter with a sequence of new ones - one per layer of the LaMA: i.e., for an n-LaMA the letter $u \in w$ is replaced by the letters $u^1, ..., u^n$ where all the u^i are different in order to have the letters belonging to different layers all distinct from each other. This renaming is always possible as the alphabet $\mathcal{U}$ is infinite. For example, for the word aba, a consistent renaming, for a 2-LaMA, could produce $a^1\, a^2\, b^1\, b^2\, a^1\, a^2$.

▶ **Definition 17** (Renaming). *$\xi^n : \mathcal{U} \to \mathcal{U}^n$ is a* renaming *function that given a letter $u \in \mathcal{U}$ generates a new sequence of n letters $u^1 \dots u^n$ with for all $i \neq j \in [1, n]$ $u^i \neq u^j$ and such that if $u_1 \neq u_2$ then for all $i, j \in [1, n]$, $u_1^i \neq u_2^j$. $\xi^n(u_1 \dots u_m) = u_1^1 \dots u_1^n \dots u_m^1 \dots u_m^n$ is its pointwise extension to words .*

Let $A = (Q, q_0, F, \Delta, V, n, M_0)$ be an n-LaMA and $w = u_1 \dots u_m \in \mathcal{U}^*$. We know that $w \in \mathcal{L}(A)$ if and only if there is a finite sequence of transitions such that for some M_f, $(q_0, M_0) \overset{w}{\underset{A}{\Longrightarrow}} (q_f, M_f)$ with $q_f \in F$. It is then possible to construct a ν-A that accepts $\xi^n(w)$, which simulates the recognition process of the n-LaMA over the word w. To do so, we encode every observable transition of A into a sequence of transitions successively simulating the constraints applied to variables of each layer. Moreover, we apply the renaming function ξ^n to the initial memory context. In order to simplify the notations, in the following, we denote by $x\lfloor_k$ the projection onto the k-th element of tuple x, e.g., $(a, b, c)\lfloor_2 = b$.

▶ **Definition 18** (Encoding of a memory context). *Let M be the memory context over the set of variables V over n layers, then $\forall v^l \in V$, its renaming through ξ^n, is defined as $[\![M]\!]_\xi(v^l) = \{\xi^n(u)\lfloor_l \mid u \in M(v^l)\}$.*

▶ **Definition 19** (Encoding of a LaMA). *Let $A = (Q, q_0, F, \Delta, V, n, M_0)$ be an n-LaMA, then the ν-A $[\![A]\!]_\xi = (Q', q_0', F', \Delta', V', M_0')$ is the encoding of A through the renaming ξ^n, where:*

- *$Q' = Q \cup Q_o$ and the set of states $Q_o = \{q_\delta^l \mid \delta = (q, \alpha, q') \in \Delta, l \in [2, n]\}$ is used by the sequence of transitions simulating each observable transition of A, $q_0{}' = q_0$ and $F' = F$;*
- *$V' = V$ is the set of variables of A flattened on one layer;*
- *$M_0{}' = [\![M_0]\!]_\xi$ is the initial memory context of A renamed in case it is not initially empty;*
- *$\Delta' = \Delta'_o \cup \Delta_\varepsilon$ where Δ_ε is the set of all non-observable transitions of A, and Δ'_o contains the encoding of every observable transition $\delta = (q, \alpha, q')$ of A, which is a sequence of transitions with $q_\delta^1 = q$ and $q_\delta^{n+1} = q'$ such that*

$$\begin{array}{rcl} \Delta'_o & = & \{(q_\delta^l, v^l, \mathbf{x}, q_\delta^{l+1}) \mid \delta = (q, \alpha, q') \in \Delta, l \in [1, n], \alpha(l) = (v^l, \mathbf{x})\} \\ & \cup & \{(q_\delta^l, \sharp, q_\delta^{l+1}) \mid \delta = (q, \alpha, q') \in \Delta, l \in [1, n], \alpha(l) = \sharp\}. \end{array}$$

Notice that the language accepted by the encoded ν-A $[\![A]\!]_\xi$ of a LaMA A is an over-approximation of the language accepted by A: $\xi(\mathcal{L}(A)) \subseteq \mathcal{L}([\![A]\!]_\xi)$. They are equal only when the LaMA has one layer (i.e., $n = 1$). Nonetheless, this construction may be used to test the membership of a word w to $\mathcal{L}(A)$. The proof is a simple induction on the length of the derivation of w and $\xi^n(w)$ [3].

▶ **Theorem 20.** *Let A be an n-LaMA. $w \in \mathcal{L}(A)$ if and only if $\xi^n(w) \in \mathcal{L}([\![A]\!]_\xi)$.*

Simulating the membership for HRA in LaMA. This section presents how to solve the membership problem for HRA using LaMA. The difference in expressiveness between HRA and LaMA comes from the ability of HRA of removing letters from histories when they are read. We resort to an encoding of words where each letter is duplicated and annotated with a number representing how many occurrences of that letter have been encountered so far. In detail, the first copy of the letter keeps the information on the number of occurrences of the letter seen so far and the second one the number of occurrences including the present one.

▶ **Example 21.** Take $w = abaca$ then the encoded word is $w' = a^0a^1\ b^0b^1\ a^1a^2\ c^0c^1\ a^2a^3$

The idea behind the encoding of observable transitions is to use the first copy to check the presence and absence of the letter in every variable (simulating the role of H_r) while the second one (that is always fresh) can be used to simulate writing and removal (hence simulating H_w). More precisely, once we add an annotated letter to a variable, the encoded automaton will ensure that the variable always stores the last seen occurrence of that letter. Thus, removing a letter from a history consists in not storing the last seen occurrence of the letter in the corresponding encoded variable. Clearly, all the letters annotated with a number smaller than the current one will not be used in any of the transitions, representing a form of garbage.

We consider a renaming function $\zeta_i : \mathcal{U} \to \mathcal{U}^2$ which replaces u by a pair of letters $u^{i-1}u^i$ for any $i \in \mathbb{N}^+$. Then, we define the encoding of words $\zeta : \mathcal{U}^* \to \mathcal{U}^*$ as follows $\zeta(u_1 \dots u_m) = \zeta_{i_{u_1}}(u_1) \dots \zeta_{i_{u_m}}(u_m)$ where each i_{u_j} is the number of occurrences of u_j seen in the prefix $u_1 \dots u_j$. Notice that when considering the word up to letter u_j, $\zeta_{i_{u_j}}(u_j)\lfloor_2$ is always a new letter (e.g., a fresh letter with respect to those in $\zeta(u_1 \dots u_j)$).

▶ **Definition 22** (Encoding of an HRA)**.** *Let $A = (Q, q_0, F, \Delta_o \cup \Delta_\varepsilon, \{h_1, \dots h_n\}, M_0)$ be an HRA, its encoding into an n-LaMA is $[\![A]\!]_\zeta = (Q', q_0', F', \Delta', V', n, M_0')$ where:*

- *$Q' = Q \cup Q_o$ and the set of states $Q_o = \{q_\delta \mid q \in Q, \delta = (q, H_r, H_w, q') \in \Delta\}$ is used by the sequence of transitions simulating each observable transition of A;*
- *$q_0' = q_0$ and $F' = F$;*
- *$V' = \{h^l, \omega^l \mid l \in [1, n]\}$ and for each layer $l \in [1, n]$, h^l plays the role of history h_l and ω^l is used to check the absence of letters in h^l.*
- *$M_0'(h^l) = \{\zeta_1(u)\lfloor_1 \mid u \in M_0(h_l)\}$ and $M_0'(\omega^l) = \emptyset$ for all $l \in [1, n]$ meaning that M_0' is as M_0 with all letters renamed with ζ_1 and empty for all extra variables;*
- *$\Delta' = \Delta_\varepsilon' \cup \Delta_o'$ with*
  - *$\Delta_\varepsilon' = \{(q, \{h^l \mid h_l \in H_\emptyset\}, q') \mid (q, H_\emptyset, q') \in \Delta_\varepsilon\}$, is the direct translation of the ε-transitions in A.*
  - *$\Delta_o' = \{(q, \alpha_{H_r}, q_\delta), (q_\delta, \alpha_{H_w}, q') \mid \delta = (q, H_r, H_w, q') \in \Delta_o\}$ with for all $l \in [1, n]$*

$$\alpha_{H_r}(l) = \begin{cases} (h^l, \mathbf{r}) & \textit{if } h_l \in H_r \\ (\omega^l, \mathbf{w}) & \textit{if } h_l \notin H_r \end{cases} \quad \textit{and} \quad \alpha_{H_w}(l) = \begin{cases} (h^l, \mathbf{w}) & \textit{if } h_l \in H_w \\ \sharp & \textit{if } h_l \notin H_w \end{cases}$$

the first simulating the guard part of the observable transition and the second the writing/relocation.

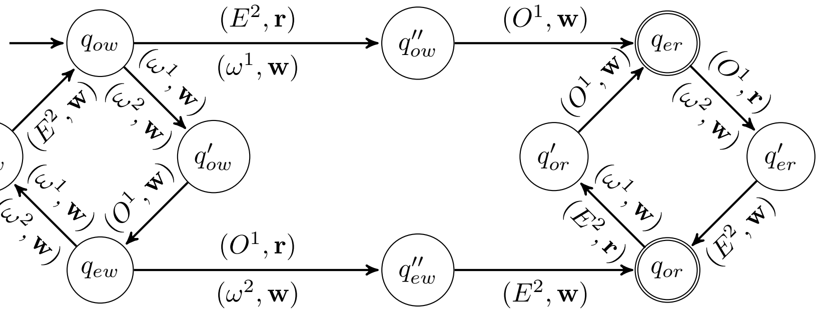


Figure 4 The 2-LaMA $\llbracket A_h \rrbracket_\zeta$, encoding of the HRA A_h from Example 16.

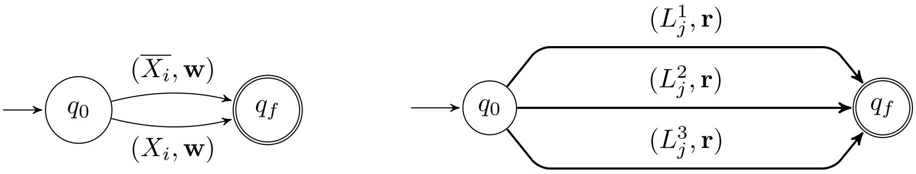
▶ **Example 23.** Fig. 4 depicts the encoding applied to the HRA of Example 16. Given the word $w = abcabb$ and its renaming $\zeta(w) = a^0a^1b^0b^1c^0c^1a^1a^2b^1b^2b^2b^3$, we present how the encoding works. A_h has two histories: $H = \{O, E\}$, thus the set of variables V of $\llbracket A_h \rrbracket_\zeta$ is $\{O^1, \omega^1\}$ on layer 1 and $\{E^2, \omega^2\}$ on layer 2. Let $(q_{ow}, M_\emptyset)$ be the initial state for both automata, with $M_\emptyset$ the memory context where all variables/histories are empty.

When reading the first letter a in A_h, only transition $(q_{ow}, M_\emptyset) \xrightarrow[A_h]{a} (q_{ew}, M_1)$ is enabled as a is not stored in any of the histories in $M_\emptyset$, as a consequence, a is added to O in M_1. In $\llbracket A_h \rrbracket_\zeta$, this transition is encoded with the sequence $(q_{ow}, M_\emptyset) \xrightarrow[\llbracket A_h \rrbracket_\zeta]{a^0} (q'_{ow}, M') \xrightarrow[\llbracket A_h \rrbracket_\zeta]{a^1} (q_{ew}, M'_1)$. The first transition when reading a^0, checks if a^0 is absent from both O^1 and E^2 using ω^1 and ω^2 with the injectivity constraint. When reading a^1 the transition $q'_{ow} \to q_{ew}$ writes the letter in O^1. Note that a^0 is still stored in ω^1 and ω^2, but it will never be read again (as the renaming ζ always increases the index of letters).

Then, when A_h read the first occurrence of b, the only enabled transition is $(q_{ew}, M_1) \xrightarrow[A_h]{b} (q_{ow}, M_2)$, where b is stored in E is M_2. And when reading c the only transition enabled is $(q_{ow}, M_2) \xrightarrow[A_h]{c} (q_{ew}, M_3)$ with O storing both a and c while E only stores b. In $\llbracket A_h \rrbracket_\zeta$, this sequence of transitions is encoded by enabling the sequence of transitions $(q_{ew}, M'_1) \xrightarrow[\llbracket A_h \rrbracket_\zeta]{b^0} (q'_{ew}, M''_1) \xrightarrow[\llbracket A_h \rrbracket_\zeta]{b^1} (q_{ow}, M'_2) \xrightarrow[\llbracket A_h \rrbracket_\zeta]{c^0} (q'_{ow}, M''_2) \xrightarrow[\llbracket A_h \rrbracket_\zeta]{c^1} (q_{ew}, M'_3)$. With M'_2 storing b^1 in E^2 and M'_3 storing c^1 in O^3 in addition to a^1. This is the only sequence of transition that can be enabled as b_0 was not stored in O^1 in the state (q_{ew}, M'_1) and c^0 was not stored in E^2 in (q_{ow}, M'_2).

When reading the second occurrence of a, the only enabled transition is $(q_{ew}, M_3) \xrightarrow[A_h]{a} (q_{or}, M_4)$ where a is transferred from O to E in M_4. In $\llbracket A_h \rrbracket_\zeta$ this is encoded by the sequence of transitions $(q_{ew}, M'_3) \xrightarrow[\llbracket A_h \rrbracket_\zeta]{a^1} (q''_{ew}, M''_3) \xrightarrow[\llbracket A_h \rrbracket_\zeta]{a^2} (q_{or}, M'_4)$. The first transition is the only one enabled in configuration (q_{ew}, M'_3) as a^1 is already stored in O^1, thus it would be impossible to write it in ω^1 to enable the transition to q'_{ew}. In M'_4, the letter a^2 is stored in E^2 along with b^1, while a^1 is still stored in O^1 but will never be read again in $\zeta(w)$, so it can be ignored. This is how the transfer mechanism is encoded in this construction.

Reading bb, the last two letters of w, will enable in A_h the sequence $(q_{or}, M_4) \xrightarrow[A_h]{b} (q_{er}, M_5)$ transferring b from E to O in M_5 and then enabling $(q_{er}, M_5) \xrightarrow[A_h]{b} (q_{or}, M_6)$ transferring b back from O to E in M_6. In $\llbracket A_h \rrbracket_\zeta$, this is encoded by reading the letters $b^1b^2b^2b^3$ and



(a) The gadget for the existentially quantified variable x_i.

(b) The 3SAT clause gadget for $C_j = (L_j^1 \vee L_j^2 \vee L_j^3)$.

Figure 5 Gadgets used for showing NP-hardness of the membership problem.

enabling the loop of transition between states q_{or}, q'_{or}, q_{er} and q'_{er}. Looking if the previous occurrence of b, here b^2 (resp. b^3), is stored in E^2 (resp. O^1) by reading in the variable. Also checking if it is absent from O^1 (resp. E^2) by writing in the ω of the same layer. Then writing the next occurrence of b, here b^3 (resp. b^4), in O^1 (resp. E^2) to encode its transfer.

Notice that, as before, the language recognized by $[\![A]\!]_\zeta$ is actually larger than $\mathcal{L}(A)$.

▶ Remark 24. In [11], HRA are presented with a set of registers able to store only one letter at a time. Their content is overwritten whenever a letter is written into it. The authors proved that HRA using only histories are as expressive as the ones using both histories and register. However, the construction presented to remove registers is exponential in their number. This is caused by the need of decoupling the overwriting into two phases, first, one uses the content to verify if an observable transition is enabled and then erases the content of histories. The exponential construction comes from the fact that to keep the languages equivalent, for each phase, one can use only one observable transition. Instead, to show membership we do not need to prove the equivalence of languages and the construction in Definition 22, already splits transitions into these two phases, using two observable transitions. Hence, it can be extended to registers avoiding the exponential cost.

▶ **Theorem 25.** *Let A be an HRA. $w \in \mathcal{L}(A)$ if and only if $\zeta(w) \in \mathcal{L}([\![A]\!]_\zeta)$.*

Complexity. The two previous encodings give polynomial reductions of the membership problem from HRA to LaMA and from LaMA to ν-A. Therefore, there is a polynomial reduction of the problem for HRA to ν-A. The expressiveness results from [4] give a linear construction from ν-A to LaMA and an exponential construction, in the number of layers, from LaMA to HRA. As ν-A are 1-LaMA, the same construction can be used to translate a ν-A into an HRA of polynomial size. This implies an equivalence of complexity class of the membership problem for ν-A and HRA, as well as for ν-A and LaMA. By transitivity, we get the same equivalence between LaMA and HRA. Next, we show that the membership problem for LaMA is NP-complete. For the hardness part, this is shown by resorting to a reduction from the 3SAT problem, while the completeness part follows by observing what would be the cost of executing a word on an automaton. Fig. 5 depicts the intuition behind the encoding of a 3SAT instance. The idea is that the gadget in Fig. 5a chooses non-deterministically the truth assignment of X_i or $\overline{X_i}$ and the one in Fig. 5b checks that this assignment indeed satisfies the given clauses.

▶ **Theorem 26.** *The membership problem for LaMA is NP-complete.*

Hence we can conclude that:

▶ **Corollary 27.** *The membership problems for ν-A, LaMA and HRA is NP-complete.*

As a direct consequence and looking at the expressiveness hierarchy in Fig. 1 we can also give a complexity class for the membership problem in FRA. Indeed, since FMA can be encoded into FRA [19], we can deduce NP-hardness, and completeness follows from their encoding into LaMA [4].

▶ **Corollary 28.** *The membership problem for FRA is NP-complete.*

4 Complexity of the non-emptiness problem

The non-emptiness problem consists in deciding whether the language accepted by an automaton is non-empty, or in other words checking if there is a path from the initial configuration to a final configuration. As mentioned before, in [11], it has been shown that deciding the non-emptiness for HRA is Ackermann-complete. Still, the complexity for non-emptiness is known neither for LaMA nor for ν-A. We start with the non-emptiness Problem for ν-A. We show that the problem is PSPACE-complete. To do so, we reduce the TQBF problem (true fully quantified Boolean formula) to ν-A non-emptiness. TQBF is known to be PSPACE-complete (Meyer-Stockmeyer theorem [1]).

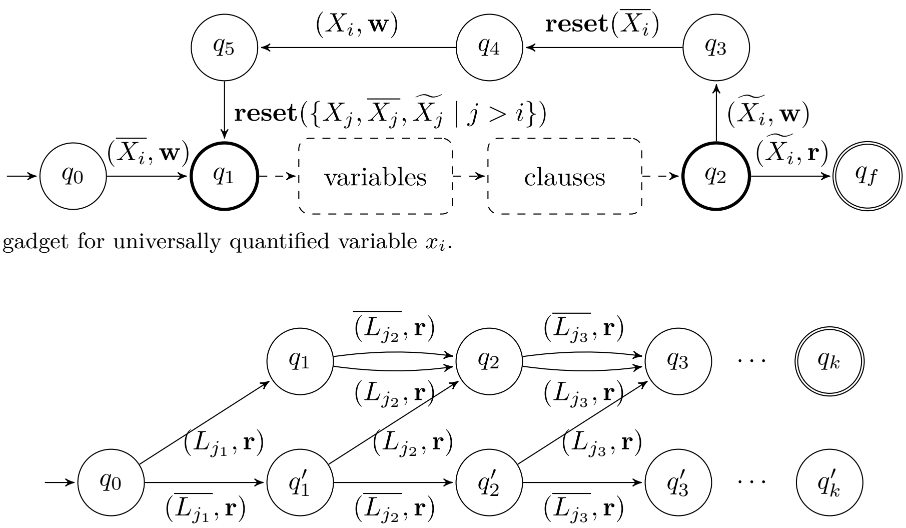
▶ **Lemma 29.** *The non-emptiness problem is PSPACE-hard for ν-A.*

Proof. Let νNEP be the short for non-emptiness Problem for ν-A.

We show that TQBF can be reduced to νNEP. Let $Q_1x_1 \dots Q_nx_n(C_1 \wedge \dots \wedge C_m)$, be a fully quantified Boolean formula, where each $Q_i \in \{\forall, \exists\}$ and each C_j is a clause comprising at most n literals (x_i or $\overline{x_i}$). We assume that literals in clauses are ordered according to the order of variable declarations and at most one literal per variable is present. To encode TQBF in ν-A we consider:

- for each existentially quantified x_i, variables X_i and $\overline{X_i}$, and the gadget depicted in Fig. 5a, used in the proof of Theorem 26;
- for each universally quantified x_i, variables X_i, $\overline{X_i}$ and $\widetilde{X_i}$, and the gadget depicted in Fig. 6a. Variable $\widetilde{X_i}$ is used as a flag to indicate that all possible truth assignments of x_i have been considered. The initial transition of the gadget initializes variable $\overline{X_i}$ to 1_i. The dashed automaton connected between states q_1 and q_2, handling other variables and the clauses, is constructed recursively. The looping part starting in state q_2 writes letter 2_i into variable $\widetilde{X_i}$, which is used after browsing once again the dashed part to reach the final state q_f. After this, variable $\overline{X_i}$ is reset and then variable X_i is initialized to 1_i to consider the other truth assignment of x_i. From state q_5 to q_1 all the variables for x_j, with j from $i+1$ to n are reset to reinitialize their truth assignments;
- for each clause C_j, a clause gadget depicted in Fig. 6b. It tests literals one after the other and takes the oblique transition for the first which makes the clause satisfied, which means that the remaining literals are just read up to the end of the clause, which is satisfied if q_f is reached.

In order to construct the ν-A A encoding the instance of TQBF we connect first the clause gadgets by merging the final state of a clause gadget with the initial state of the next one. Let C be the resulting automaton. Then, we connect to C the gadgets for variable declarations starting from the nth, i.e., the last in the order of declarations. If the variable is under an existential quantifier, we connect the existential variable gadget in front of the automaton obtained so far by merging its final state with the initial state of C. If the variable is under a universal quantifier, we connect the corresponding gadget by merging the initial state of C with state q_1 of the gadget, and the final state of C with state q_2 of the gadget.


q_5
$(X_i, \mathbf{w})$
q_4
$\mathbf{reset}(\overline{X_i})$
q_3
$\mathbf{reset}(\{X_j, \overline{X_j}, \widetilde{X_j} \mid j > i\})$
$(\widetilde{X_i}, \mathbf{w})$
q_0
$(\overline{X_i}, \mathbf{w})$
q_1
variables
clauses
q_2
$(\widetilde{X_i}, \mathbf{r})$
q_f
q_1
$(\overline{L_{j_2}}, \mathbf{r})$
q_2
$(\overline{L_{j_3}}, \mathbf{r})$
q_3
$\cdots$
q_k
$(L_{j_2}, \mathbf{r})$
$(L_{j_3}, \mathbf{r})$
$(L_{j_1}, \mathbf{r})$
$(L_{j_2}, \mathbf{r})$
$(L_{j_3}, \mathbf{r})$
q_0
q'_1
q'_2
q'_3
$\cdots$
q'_k
$(\overline{L_{j_1}}, \mathbf{r})$
$(\overline{L_{j_2}}, \mathbf{r})$
$(\overline{L_{j_3}}, \mathbf{r})$

(a) The gadget for universally quantified variable x_i.

(b) The gadget for TQBF clause $C_j = (L_{j_1} \vee \ldots \vee L_{j_k})$.

Figure 6 Gadgets used for showing NP-hardness of the emptiness problem for ν-A.

We connect this way, i.e., following the inverse order of declarations, the gadgets for all the remaining variable declarations. The initial state of the first declared variable gadget is the initial state of A and the unique state q_f of the final construction is the unique final state of A. Finally, the input word w is obtained recursively for each TQBF instance by the function $\mathit{input}(\phi)$ defined in Algorithm 1. The construction of the word follows the intuition given above (for the construction of the automaton), that is: it unfolds the loops generating the letters needed at each step.

Algorithm 1 Function to generate the word accepted by TQBF automaton.

```
function INPUT($\phi$)                                   ▷ $\phi = Q_1 x_1 \ldots Q_n x_n \; C_1 \ldots C_m$
    $\forall i \in [1, n] : \mathit{init}(x_i) = 1_i$
    $\forall i \in [1, n] : \mathit{end}(x_i) = 2_i$
    $\forall j \in [1, m] : w_j$          ▷ contains exactly one $1_i$ for each $x_i$ or $\overline{x_i}$ present in clause $C_j$
    if $\phi = \emptyset$ then return $\epsilon$
    else if $\phi = \exists x_i \; \phi'$ then return $\mathit{init}(x_i).\mathit{input}(\phi')$
    else if $\phi = \forall x_i \; \phi'$ then return $\mathit{init}(x_i).\mathit{input}(\phi').\mathit{end}(x_i).\mathit{init}(x_i).\mathit{input}(\phi').\mathit{end}(x_i)$
    else if $\phi = C_i \; \phi'$ then return $w_i.\mathit{input}(\phi')$
```

Example: for the TQBF instance $\exists x_1 \; \forall x_2 \; \forall x_3 \; \exists x_4((x_1 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_4))$, Fig. 7 represents a general construction schema of the corresponding ν-A, and the input word is $1_1 1_2 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 2_2 1_2 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 1_2 1_3 1_4 1_1 1_4 1_2 1_3 1_4 2_3 2_2$.

Note that every gadget of the automaton is deterministic, except for the existential variable gadget. The size of A is polynomial in the size of the TQBF expression. The length of the word generated by Algorithm 1 is in $\Omega(2^n)$ but it is not a parameter of the construction of A. Clearly, only a word generated by Algorithm 1 (or a consistent renaming) can be accepted by A starting with an empty memory context. Such a word can be accepted if and only if there is a solution to the TQBF instance. ◀

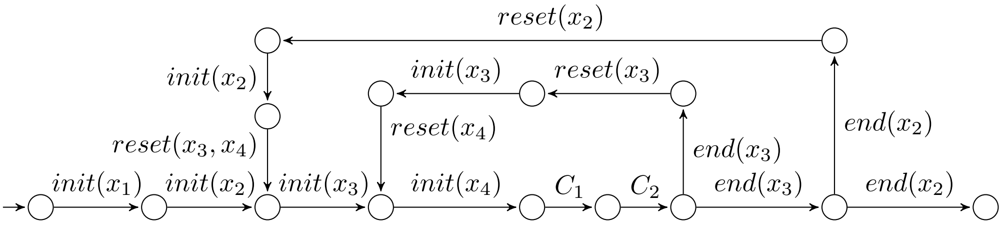


Figure 7 Schema of construction for TQBF instance $\exists x_1\ \forall x_2\ \forall x_3\ \exists x_4\ ((x_1 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_4))$.

It remains to show that the non-emptiness problem for ν-A is in PSPACE. This accounts for showing that if the language recognized by an ν-A is non-empty then it contains a word whose size together with the length of the transition path needed to accept it, are exponentially bounded with respect to the size of the ν-A. To this aim, we "build" a finite state machine (FSM) characterizing an abstraction of the state space of the ν-A.

If one can choose the letters to read, the idea is that observable transitions that write a letter in a variable are never blocking. Since the alphabet is infinite there is always a fresh letter that can be added, which we call a token. Instead, observable transitions that read a letter from a variable are blocking, in the sense that concerned variables must contain at least a letter (that we call a key). The first step towards the construction of the FSM is to build a *canonical* ν-A such that a word accepted by the canonical automaton will also be accepted by the initial ν-A A and each word accepted by A will have a corresponding canonical version. Consider a ν-A $A = (Q, q_0, F, \Delta, V, M_0)$, its *canonical* version $cano(A) = (Q, q_0, F, \Delta, V, M'_0)$ is an ν-A over the alphabets K and T, where:

- $K \subset \mathcal{U}$, such that $|K| = |V|$, is the set of *keys* k_v, each of them being associated with a variable $v \in V$. If $M_0(v) \neq \emptyset$, we select k_v in $M_0(v)$. Also, if $M_0(v) = \emptyset$, we select k_v such that $\forall v' \in V, k_v \notin M_0(v')$. The presence of a key in a variable v denotes the fact that v is non-empty.
- $T = \{t_1, t_2, \ldots\} \subset \mathcal{U}$, $K \cap T = \emptyset$, is an infinite set containing letters called *tokens* intended to be used only once, which are never stored in memory. Hence, no letter in T is present in the initial memory context of A, $\forall v \in V : M_0(v) \cap T = \emptyset$.
- For each $v \in V$, the initial memory context $M'_0(v)$ of $cano(A)$ is either empty if $M_0(v) = \emptyset$, or if $M_0(v) \neq \emptyset$, it only contains its key k_v.

Notice that a word w is accepted by $cano(A)$ if the following conditions hold:

1. $w \in (K \cup T)^*$, and if $t_i \in T$ appears in w then it occurs at most once,
2. let $(q_0, M'_0) \xRightarrow[cano(A)]{w} (q_f, M_f)$ with $q_f \in F$ be the accepting path for w then for each intermediate configuration (q, M) in the path and for each $k_v \in K$ either $k_v \in M(v)$ or for all $v' \in V$, $k_v \notin M(v')$.

Observe that $cano(A)$ is actually the same automaton as A but over a subset of the alphabet $\mathcal{U}$ and where for each $v \in V$, $M'_0(v) \subseteq M_0(v)$, hence it is easy to conclude that the language of $cano(A)$ is included in the one of A.

▶ **Lemma 30.** *Let A and $cano(A)$ be a ν-A and its canonical version. If a word $w \in \mathcal{L}(cano(A))$ then $w \in \mathcal{L}(A)$.*

We want to show that the language accepted by a ν-A A is empty if and only if the language accepted by $cano(A)$ is empty. The if part is the most involved and is the content of the following lemma.

▶ **Lemma 31.** *Let A be a ν-A and $cano(A)$ and its canonical version. If $w \in \mathcal{L}(A)$ then there exists $w' \in \mathcal{L}(cano(A))$.*

Proof. Let $w = u_1 \dots u_n \in \mathcal{L}(A)$. Then there exists an accepting path $(q_0, M'_0) \overset{w}{\underset{A}{\Longrightarrow}} (q_f, M_f)$ with $q_f \in F$ and intermediate configurations $(q_i, M_i)_{i \geq 0}$. Depending on those intermediate configurations we build a new word $w' = u'_1 \dots u'_n$ and the corresponding path in $cano(A)$ accepting w'. For each configuration (q_i, M_i), the construction maintains an invariant: $\forall v \in V, M'_i(v) = \{k_v\}$ if and only if $M_i(v) \neq \emptyset$. The proof proceeds by induction:
Base case: By construction the initial configuration of $cano(A)$ satisfies the invariant.
Inductive step: We examine the transition $(q_i, M_i) \overset{\delta_i}{\underset{u_i}{\longrightarrow}} (q_{i+1}, M_{i+1})$. By inductive hypothesis, we know that there exists a sequence of transitions $(q_0, M'_0) \overset{u'_1 \dots u'_{i-1}}{\underset{cano(A)}{\Longrightarrow}} (q_i, M'_i)$ such that $\forall v \in V, M'_i(v) = \{k_v\}$ if and only if $M_i(v) \neq \emptyset$. We prove there is a letter u'_i leading to the configuration (q_{i+1}, M'_{i+1}) satisfying this property, through δ_i (by construction A and $cano(A)$ are defined on the same set of transitions), we list all possible cases:

- $\boldsymbol{\delta_i = (q_i, \text{reset}, q_{i+1})}$: then $u_i = u'_i = \varepsilon$ and δ_i will lead to a configuration with $M'_{i+1}(v) = \emptyset$ if $v \in$ **reset** or $M'_{i+1}(v) = M'_i(v)$ otherwise. Hence satisfying the invariant.
- $\boldsymbol{\delta_i = (q_i, v, \text{r}, q_{i+1})}$: then $M_i(v) \neq \emptyset$ otherwise the transition could not be enabled, so $u'_i = k_v$ and by inductive hypothesis $M'_i(v) = \{k_v\}$. Since the memory context does not change for both automata, the invariant is satisfied;
- $\boldsymbol{\delta_i = (q_i, v, \text{w}, q_{i+1})}$: if $M_i(v) = \emptyset$, then $u'_i = k_v$, and k_v will be written in variable v in M'_{i+1} satisfying the invariant.
  If $M_i(v) \neq \emptyset$, then $u'_i = t_i \in T$ is a token and $M'_i(v) = M'_{i+1}(v)$ since tokens are not stored in memory. By inductive hypothesis we know that $M'_i(v) = \{k_v\}$ and as δ_i is a writing transition, then $M_{i+1} \neq \emptyset$, satisfying the invariant.

From the previous construction, it follows immediately that $w' \in \mathcal{L}(cano(A))$. ◀

Observe that, when reading a word $w \in \mathcal{L}(cano(A))$, we only need to store the letters belonging to K. Indeed, tokens in T may occur only once in w. This entails that tokens can only enable a write observable transition, while for read transitions keys are sufficient. Hence, in practice, tokens do not need to be added to the memory context. Hence the number of different configurations in $cano(A)$ is bounded by $|Q| \cdot 2^{|K|}$ as:

- we have $|Q|$ states that can be encoded on $log|Q|$ bits, and
- there are $2^{|K|}$ possibilities to store the presence or not in the memory of letters in K (2 possibilities per letter encoded on 1 bit since each k_v can only be stored in v), so in total we need $|K|$ bits.

This shows that the number of configurations is finite. On top of this, as remarked above, transitions over letters in T do not add constraints on the memory context and they can be ignored. Hence the alphabet is now finite and we can reduce the non-emptiness of FSM to the non-emptiness problem of ν-A.

▶ **Lemma 32.** *The non-emptiness problem for ν-A is in PSPACE*

Proof. Given a ν-A $A = (Q, q_0, F, \Delta, V, M_0)$, its canonical form has at most $|Q|2^{|K|}$ configurations. The state space of $cano(A)$ could be constructed as an FSM by merging all transitions of A writing a token from T going from state q to q' into a unique transition. This way, the FSM would have $O(|\Delta|2^{|K|})$ transitions as each configuration (q, M') of $cano(A)$ has at most as many outgoing transitions as q in A. A formal definition of the construction is in [3].

Moreover, if the underlying FSM is non-empty it implies that there is a sequence of at most $O(|\Delta|2^{|K|})$ transitions from an initial state of A to one of its accepting state. Recall that finding a path between two vertexes/states in a graph (V, E) is a problem called PATH which is NL-complete [1]. The algorithm for PATH, in logarithmic space, could be adapted to find whether there exists a sequence of transitions from an initial state of A to an accepting state. Since this sequence of transitions is exponential in the size of A, we prove that the problem is in NPSPACE for ν-A. Since PSPACE=NPSPACE [1] we show that the non-emptiness problem for ν-A is in PSPACE.

The PATH algorithm adapted to our problem memorizes a state of A, the memory context of $cano(A)$ and a counter on $O(\log(|\Delta|) + |K|)$ bits. Each time that the counter is augmented by one, a transition starting in the memorized state will be chosen randomly and applied as follows: if this transition is a reset, then the state is updated and the memory is reset. If this transition is a write, then the state is updated and the corresponding key is added to the memory (if not already present). If the transition is a read then either the key is not in the memory and the algorithm halts and rejects or the state is updated. As soon as an accepting state is reached then the algorithm halts and accepts. If the counter reaches its maximum then the algorithm halts and rejects. Note that the FSM is not actually constructed in this algorithm, but only one of its paths is explored dynamically. ◀

▶ **Theorem 33.** *The non-emptiness problem for ν-A PSPACE-complete.*

Proof. By Lemmata 29 and 32. ◀

5 Conclusions

We have discussed the complexity of membership and non-emptiness for three formalisms ν-A, LaMA and HRA. We showed that concerning the membership problem, all three kinds of automata fall in the NP-complete class. Non-emptiness is more delicate. We proved that the non-emptiness problem for ν-A is PSPACE-complete.

For LaMA, we know the lower bound and the upper bound of the complexity class of the non-emptiness problem. As a consequence of Theorem 33 and from the expressiveness results in [4], the complexity is PSPACE-hard. However, it is a strict lower bound as we are able to construct a LaMA where the shortest accepted word is of size in $O(2^{2^n})$ with n the number of variables. In our previous work [4], we showed an exponential encoding of LaMA into HRA for which the non-emptiness problem is shown to be Ackermann-complete in [11]. This also gives us the Ackermann class membership. As one of the reviewers suggested, we believe that we could adapt to LaMA the proof in [11] to show that the problem is actually Ackermann-complete.

As for future work, apart from formally showing the Ackermann-completeness of the non-emptiness problem for LaMA, we plan to address other expressiveness issues of LaMA. Indeed the number of layers seems to create a hierarchy of expressiveness and complexity.

References

1 S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2006. URL: https://theory.cs.princeton.edu/complexity/book.pdf.

2 Clement Bertrand. *Reconnaissance de motifs dynamiques par automates temporisés à mémoire.* Theses, Université Paris-Saclay, December 2020. URL: https://theses.hal.science/tel-03172600.

3 Clément Bertrand, Cinzia Di Giusto, Hanna Klaudel, and Damien Regnault. Complexity of Membership and Non-Emptiness Problems in Unbounded Memory Automata. Technical report, Université Côte d'Azur, CNRS, I3S, France ; IBISC, Univ. Evry, Université Paris-Saclay, France ; Scalian Digital Systems, Valbonne, France, 2023. URL: `https://hal.science/hal-04155339`.

4 Clément Bertrand, Hanna Klaudel, and Frédéric Peschanski. Layered memory automata: Recognizers for quasi-regular languages with unbounded memory. In Luca Bernardinello and Laure Petrucci, editors, *Application and Theory of Petri Nets and Concurrency - 43rd International Conference, PETRI NETS 2022, Bergen, Norway, June 19-24, 2022, Proceedings*, volume 13288 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2022. `doi:10.1007/978-3-031-06653-5_3`.

5 Clément Bertrand, Frédéric Peschanski, Hanna Klaudel, and Matthieu Latapy. Pattern matching in link streams: Timed-automata with finite memory. *Sci. Ann. Comput. Sci.*, 28(2):161–198, 2018. URL: `http://www.info.uaic.ro/bin/Annals/Article?v=XXVIII2&a=1`.

6 Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4):702–715, 2010. Fundamentals of Computation Theory. `doi:10.1016/j.tcs.2009.10.009`.

7 Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011. `doi:10.1145/1970398.1970403`.

8 Aurélien Deharbe. *Analyse de ressources pour les systèmes concurrents dynamiques*. PhD thesis, Université Pierre et Marie Curie, France, September 2016. URL: `https://tel.archives-ouvertes.fr/tel-01523979`.

9 Aurelien Deharbe and Frédéric Peschanski. The omniscient garbage collector: A resource analysis framework. In *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*, pages 102–111. IEEE Computer Society, 2014. `doi:10.1109/ACSD.2014.18`.

10 Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3), April 2009. `doi:10.1145/1507244.1507246`.

11 Radu Grigore and Nikos Tzevelekos. History-register automata. *Log. Methods Comput. Sci.*, 12(1), 2016. `doi:10.2168/LMCS-12(1:7)2016`.

12 Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*, volume 6031 of *Lecture Notes in Computer Science*, pages 561–572. Springer, 2010. `doi:10.1007/978-3-642-13089-2_47`.

13 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. `doi:10.1016/0304-3975(94)90242-9`.

14 Michael Kaminski and Daniel Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5):741–760, 2010. `doi:10.1142/S0129054110007532`.

15 Ahmet Kara. *Logics on data words: Expressivity, satisfiability, model checking*. PhD thesis, Technical University of Dortmund, Germany, 2016. URL: `http://hdl.handle.net/2003/35216`.

16 Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Polynomial-time equivalence testing for deterministic fresh-register automata. In Igor Potapov, Paul G. Spirakis, and James Worrell, editors, *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*, volume 117 of *LIPIcs*, pages 72:1–72:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.MFCS.2018.72`.

17 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004. `doi:10.1145/1013560.1013562`.

18 Hiroshi Sakamoto and Daisuke Ikeda. Intractability of decision problems for finite-memory automata. *Theoretical Computer Science*, 231(2):297–308, 2000. doi:10.1016/S0304-3975(99)00105-X.

19 Nikos Tzevelekos. Fresh-register automata. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 295–306. ACM, 2011. doi:10.1145/1926385.1926420.

Modal Logics for Mobile Processes Revisited

Tiange Liu
School of Computing, The Australian National University, Canberra, Ngunnawal Country, Australia

Alwen Tiu
School of Computing, The Australian National University, Canberra, Ngunnawal Country, Australia

Jim de Groot
School of Computing, The Australian National University, Canberra, Ngunnawal Country, Australia

Abstract

We revisit the logical characterisations of various bisimilarity relations for the finite fragment of the π-calculus. Our starting point is the early and the late bisimilarity, first defined in the seminal work of Milner, Parrow and Walker, who also proved their characterisations in fragments of a modal logic (which we refer to as the MPW logic). Two important refinements of early and late bisimilarity, called open and quasi-open bisimilarity, respectively, were subsequently proposed by Sangiorgi and Walker. Horne, et. al., showed that open and quasi-bisimilarity are characterised by intuitionistic modal logics: OM (for open bisimilarity) and FM (for quasi-open bisimilarity). In this work, we attempt to unify the logical characterisations of these bisimilarity relations, showing that they can be characterised by different sublogics of a unifying logic. A key insight to this unification derives from a reformulation of the four bisimilarity relations (early, late, open and quasi-open) that uses an explicit name context, and an observation that these relations can be distinguished by the relative scoping of names and their instantiations in the name context. This name context and name substitution then give rise to an accessibility relation in the underlying Kripke semantics of our logic, that is captured logically by an S4-like modal operator. We then show that the MPW, the OM and the FM logics can be embedded into fragments of our unifying classical modal logic. In the case of OM and FM, the embedding uses the fact that intuitionistic implication can be encoded in modal logic $S4$.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Modal and temporal logics

Keywords and phrases pi-calculus, modal logic, intuitionistic logic, bisimilarity

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.34

Acknowledgements We thank Ross Horne for various insightful discussions on the subtlety of the mismatch operator, and the anonymous reviewers for their careful reading of the paper and their helpful feedback on an earlier version of this paper.

1 Introduction

The π-calculus [14] is a process calculus originally developed by Milner, Parrow and Walker, aimed at modelling a notion of process mobility (called link mobility). It can be seen as an extension of the Calculus of Communicating Systems (CCS) [13], that allows the creation of channel names, and exchanges of names between processes. Unlike CCS, where there is a canonical notion of (strong/weak) bisimilarity defining process equivalence, there are several notions of (strong/weak) bisimilarity for the π-calculus that arise from different ways in which name quantification is scoped in the bisimulation game. We consider four important notions of bisimilarity in this work: the early and the late bisimilarity, that were first defined in [14], the open bisimilarity [20] and the quasi-open bisimilarity [22]. The latter two are chosen for our study for two reasons: they are full congruence relations (closed under all process constructs, something which is not true for early/late bisimilarity), and they are more amenable for automation, especially open bisimilarity. Quasi-open bisimilarity implies early bisimilarity and is implied by open bisimilarity, but is incomparable to late bisimilarity.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 34; pp. 34:1–34:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our main interest is in the problem of characterising bisimilarity using (modal) logic. For a modal logic to characterise a notion of bisimilarity, bisimilar processes should satisfy precisely the same formulas (soundness), and conversely two processes satisfying the same formulas should be bisimilar (completeness). This type of results was pioneered by Hennessy and Milner, who characterised bisimilarity for CCS using a classical normal multi-modal logic [9]. Modal logics characterising early and late bisimilarity for the π-calculus were developed by Milner, Parrow and Walker [15]. The authors of that work show that early and late bisimilarity are characterised by the modal logic $\mathbb{EM}$ and $\mathbb{LM}$, respectively. Both $\mathbb{EM}$ and $\mathbb{LM}$ are sublogics of a classical modal logic, which we refer to here as MPW logic. Such a logical characterisation for open and quasi-open bisimilarity remained open until recently. In 2017, a characterisation of open bisimilarity was given using an intuitionistic modal logic $\mathbb{OM}$ [4, 5]. The intuitionistic nature of their logic, as opposed to MPW classical modal logic, was motivated by the fact that closure under certain name substitutions acts like intuitionistic persistence. Not long after, quasi-open bisimilarity was characterised using yet another intuitionistic modal logic called $\mathbb{FM}$ [10].

Early and late bisimilarity are distinguished in one important case involving an input transition. In a bisimulation game between a process P and another process Q, if P makes an input transition, e.g., $P \xrightarrow{a(x)} P'(x)$, then the move that Q plays can depend, or not depend, on the choice of the name x. In early bisimulation, the choice that Q makes is dependent on x, whereas in late bisimulation, it is independent of x. In open bisimulation, which refines late bisimulation, the choice of x is further delayed indefinitely – technically this is formalised by allowing input names to be *instantiated* at any point in the bisimulation game. In quasi-open bisimulation, the choice that Q makes is dependent on the name x, just like in early bisimulation. However, like open bisimulation, input names can be further instantiated at any point in the bisimulation game. Both open and quasi-open bisimulation impose a restriction on name substitutions, permitting only substitutions that do not identify certain pairs of names (typically those arising from names generated from bound-output transitions); they differ only in the extent on how certain names must remain *distinct* throughout the bisimulation game. The open nature of name instantiations is essentially what gives both open and quasi-open bisimiliarty their intuitionistic character: in the bisimulation game, equality between two (input) names cannot generally be decided, i.e. the classical tautology $(x = y) \vee (x \neq y)$ does not necessarily hold at every point in the bisimulation game [5, 10].

While the difference between early and late bisimilarity is reflected in MPW logic by the use of two modalities that capture precisely the difference in the scope of the name quantification arising from input transitions, the same cannot be said about $\mathbb{OM}$ and $\mathbb{FM}$, at least in their current formulations in [5, 10]. An obvious reason is that $\mathbb{OM}$ and $\mathbb{FM}$ are entirely separate logics, so not sublogics of a unifying logic like MPW logic. Another is a more fundamental one: the notions of *name distinctions* used in open and quasi-open bisimilarity are quite different, at least superficially, with open bisimulation adopting a more relax notion (that allows more names to be identified). This fundamental difference gives rise to seemingly incompatible logics and it is not obvious how they can be viewed as sublogics of a unifying logic. In this work, we show that these differences can be reconciled if the *context* in which these names are instantiated is taken into account. A (name) context here refers to information about how a name is created (as part of an input or a bound output), and the relative order in which names are created in a trace of a process. We refer to such a context as a *history*. By reformulating all four bisimilarity relations by explicitly accounting for histories of names, we are able to obtain a unifying (classical) modal logic, whose sublogics characterise all four bisimilarity relations. This reconciliation needs to account for the difference between intuitionistic and classical modal logics. We achieve this by viewing name substitutions as giving rise to an accessibility relation in a Kripke model, with a corresponding modal operator $\Box$ that behaves like a modal operator in the modal logic S4 (i.e., normal modal logics with reflexive and transitive frames). We can then encode intuitionistic implication (or negation) using this modal operator and classical implication. In particular, the notion of inequality $x \neq y$ in the intuitionistic logic $\mathbb{FM}$ becomes the modal formula $\Box\neg(x = y)$ in our logic. One notable consequence of our unifying logic, in the context of quasi-open bisimilarity, is that we obtain a much simpler construction of the distinguishing formulas for processes that are not quasi-bisimilar, in comparison to [10].

Our contributions can be summarised as follows:

- We give a uniform reformulation of four bisimilarity relations (early, late, open and quasi-open) using explicit name contexts.
- We give a unifying logic, whose sublogics characterise all four bisimilarity relations mentioned above.
- We provide a new construction of distinguishing formulae for quasi-open bisimilarity, simplifying a similar construction in [10], by making essential use of classical negation.

Outline of the paper. We set the stage in Section 2, where we recall the π-calculus, its late operational semantics with respect to a history, and the definitions of late, early, open and quasi-open bisimilarity. In Section 3 we introduce the logic that lies at the heart of this paper, together with its semantics. We showcase the distinguishing power by examples, and we state the main theorem of the paper, giving four fragments of the logic each of which characterises a notion of bisimilarity. Section 4 is devoted to proving the completeness part of the main theorem. Our results currently are established for the finite π-calculus with the match operator, but without the mismatch operator. In Section 5 discuss some key ideas on how to extend our results to handle the mismatch operator, leaving the details to future work. In Section 6 we discuss related work and we conclude in Section 7. Some detailed proofs are omitted but will be made available in a forthcoming technical report.

2 π-calculus and four notions of bisimulations

We give a brief overview of the operational semantics of the finite fragment of the π-calculus [14], and reformulate four notions of bisimulation: early [14], late [14], open [20] and quasi-open [22] bisimulation.

We assume a countably infinite set of *channel names* $\mathcal{N}$, elements of which are ranged over by lower-case letters such as x, y and z. Each name x has its dual *co-names*, denoted by $\bar{x}$. Informally, a name represents a communication channel where input can be received, and a co-name represents a channel where output can be sent. Processes can synchronise along channels with complementary names, i.e., a process inputting on channel x can synchronise with another process outputting on channel $\bar{x}$.

▶ **Definition 1.** Processes are defined by the grammar

$$P ::= 0 \mid \tau.P \mid \bar{x}y.P \mid x(z).P \mid \nu x.P \mid (P \mid P) \mid P + P \mid [x = y]P.$$

A process of the form $\bar{x}y.P$ is an output-prefixed process, representing a process capable of outputting a free name y along channel x. We adopt here a syntactic sugar of the form $\bar{x}(z).P$ as an abbreviation of $\nu x.\bar{x}z.P$. Semantically, this represents a process capable of outputting a bound name z along channel x. A process of the form $x(z)$ is an input-prefixed process, with z acting as a placeholder for the received name. The prefix τ is a silent prefix, meaning that the transition can act without interaction with the environment, and $\nu x.P$ turns x into a bound name in P. The processes $P \mid Q$, $P+Q$ and $[x=y]P$ represent parallel processes, choice, and match, respectively.

▶ **Definition 2.** We recall some basic definitions.

- A name z in a prefix π is *binding* if π is of the form $\bar{x}(z)$ or $x(z)$. We write $\mathrm{bn}(\pi)$ for the binding names and $\mathrm{fn}(\pi)$ for all other names in π.
- An occurrence of a name z in a process P is *bound* if it lies in the scope of a prefix of the form $\bar{x}(z)$, $x(z)$, or of νz. Occurrences of names that are not bound are called *free*. We write $\mathrm{bn}(P)$ and $\mathrm{fn}(P)$ for the sets of bound and free names of a process, and we abbreviate $\mathrm{fn}(P,Q) = \mathrm{fn}(P) \cup \mathrm{fn}(Q)$.
- A *substitution* σ is a map that sends names to names such that the *support* $\mathsf{supp}(\sigma) := \{x \mid \sigma(x) \neq x\}$ of σ is finite. We sometimes write $\{{}^{z_1,\dots,z_n}/_{x_1,\dots,x_n}\}$ for the substitution σ with $\mathsf{supp}(\sigma) = \{x_1,\dots,x_n\}$ and $x_i\sigma = z_i$ for all $i \in \{1,\dots,n\}$. The application of a substitution to a variable x, prefix π or process P is defined as expected and denoted by $x\sigma$, $\pi\sigma$ and $P\sigma$. The composition $\sigma \cdot \theta$ of two substitutions is defined by $(\sigma \cdot \theta)(x) = \theta(\sigma(x))$.

2.1 History and operational semantics

Before defining our operational semantics of the π-calculus, we introduce the notion of a history and a respectful substitution, adapting the same notion from [20].

▶ **Definition 3** (Histories)**.** A history h is a list of names annotated with either i (denoting an input name) or o (denoting an output name). If x is any name then we write $x \in h$ if x^i or x^o appears in h, and if X is a set of names then we write $X \subseteq h$ if $x \in h$ for all $x \in X$.

When enumerating the list of annotated names in a history, we separate each name in the list with dots, e.g., $x^i \cdot y^o \cdot z^i$.

Intuitively, a history h represents the list of names that a process sends and receives during its transitions. The o-annotated names (denoted by z^o) correspond to output names extruded by a process in its bound output transitions. The names marked as input (denoted by z^i) represent symbolic inputs (i.e., variables) received by a process. The difference between these annotations is captured in the following definition of *respectful substitutions*.

▶ **Definition 4** (Respectful substitutions)**.** A substitution σ *respects* h if, for all h', h'' and x such that $h = h' \cdot x^o \cdot h''$, we have $x\sigma = x$ and $y\sigma \neq x$ for all $y \in h'$. If $h = x_1^{p_1} \cdots x_n^{p_n}$ is a history, where $p_1,\dots,p_n \in \{i,o\}$, then we let $h\sigma := (x_1\sigma)^{p_1} \cdots (x_n\sigma)^{p_n}$ be the application of a respectful substitution σ to h.

▶ **Example 5.** Let $h = a^i \cdot b^o \cdot c^o \cdot x^i \cdot y^i$. Then $\sigma_1 = \{b/x, y/a\}$ is a substitution that respects h, and applying σ_1 to h results in $h\sigma_1 = a^i \cdot b^o \cdot c^o \cdot b^i \cdot a^i$. (Notice that we allow names to be repeated in a history). On the other hand, $\sigma_2 = \{a/b\}$ is not an h-respectful substitution, as it violates the condition that o-annotated names cannot be substituted, i.e., that $b\sigma_1 = b$ fails to hold. The substitution $\sigma_3 = \{c/a\}$ also does not respect h, as it substitutes an i-annotated name a with an o-annotated name that appears later in the history.

As the above example illustrates, the o-annotated names act like constants, while i-annotated names act like scoped variables, with their scoping determined by their relative positions in the history. Intuitively, when we consider a history as a trace of names inputted and outputted by a process, this scoping enforces the fact that a name received earlier in the

$$\frac{}{h : \pi.P \xrightarrow{\pi} P}\ (\textsc{Act}) \qquad \frac{h \cdot z^o : P \xrightarrow{\pi} Q \quad z \notin h \cup \mathrm{bn}(\pi) \cup \mathrm{fn}(\pi)}{h : \nu z.P \xrightarrow{\pi} \nu z.Q}\ (\textsc{Res})$$

$$\frac{h \cdot z^o : P \xrightarrow{\bar{x}z} Q \quad z \notin \{x\} \cup h}{h : \nu z.P \xrightarrow{\bar{x}(z)} Q}\ (\textsc{Open}) \qquad \frac{h : P \xrightarrow{\bar{x}(z)} P' \quad h : Q \xrightarrow{x(z)} Q'}{h : P|Q \xrightarrow{\tau} \nu z.(P'|Q')}\ (\textsc{Close})$$

$$\frac{h : P \xrightarrow{\pi} R}{h : P + Q \xrightarrow{\pi} R}\ (\textsc{Sum}) \qquad \frac{h : P \xrightarrow{\pi} Q \quad \mathrm{bn}(\pi) \cap \mathrm{fn}(R) = \emptyset}{h : P|R \xrightarrow{\pi} Q|R}\ (\textsc{Par})$$

$$\frac{h : P \xrightarrow{\pi} R}{h : [x = y]P \xrightarrow{\pi} R}\ (\textsc{Match}) \qquad \frac{h : P \xrightarrow{\bar{x}y} P' \quad h : Q \xrightarrow{x(z)} Q'}{h : P|Q \xrightarrow{\tau} P'|Q'\{y/z\}}\ (\textsc{L-Com})$$

Figure 1 The late transition semantics of the π-calculus with histories. Their symmetric variants are omitted. We require that $\mathrm{fn}(P) \subseteq h$ whenever $h : P \xrightarrow{\pi} Q$.

trace cannot be identified with a fresh name outputted later. The meaning of the annotations of names in a history and respectful substitutions will become clearer later when we define various notions of bisimilarity (Section 2.2).

We next define two orderings that will be useful later in the definitions of bisimulation. These orderings intend to constrain the possible identification of names in a history as a result of applying a respectful substitution.

▶ **Definition 6** (Orderings on histories). *We write $h \subseteq_o h'$ if h' can be obtained from h by adding o-annotated names to the end. Similarly, we write $h \subseteq_i h'$ if h' can be obtained from h by adding i-annotated names in front of h.*

The ordering $h \subseteq_i h'$ is intended to capture the fact that the new i-annotated names in h' cannot be identified with any o-annotated names in h (but may be identified with i-annotated names) after applying an h'-respectful substitution. This fact will be important later when defining quasi-open bisimulation. In the ordering $h \subseteq_o h'$ the new o-annotated names cannot be identified with any names appearing in h. This will be used later in the definition of early- and late-bisimulation.

The operational semantics of the π-calculus is given in Figure 1. Note that we use the *late variant* of the semantics [14], where bound input is not instantiated directly in the transition relation; its instantiation is defined in the definitions of bisimulation (see Section 2.2). Our semantics differs slightly from the standard late transition semantics, as each transition is indexed by a history. The history is strictly speaking not needed for the semantics in Figure 1. However, it will be important later when we discuss the handling of the mismatch operator (see Section 5). Note that in the Open and Res, the ν-binder in the process expression is interpreted as an o-annotated name in the history, reflecting the fact that this name is not affected by respectful substitutions.

We now state several lemmas for future reference.

▶ **Lemma 7.** *Let h be a history and suppose σ, θ are substitutions such that σ respects h. Then θ respects $h\sigma$ if and only if $\sigma \cdot \theta$ respects h.*

▶ **Lemma 8.** *Suppose $h : P \xrightarrow{\pi} Q$. If π is of the form τ or $\bar{x}y$ then $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P)$. If π is of the form $x(z)$ or $\bar{x}(z)$ then $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P) \cup \{z\}$.*

The following lemma helps prove that (quasi-)open bisimulations are closed under respectful substitutions. In a given transition $h : P \xrightarrow{\pi} Q$, without loss of generality, we may assume that $\mathrm{bn}(\pi)$ are chosen to be sufficiently fresh.

▶ **Lemma 9** (monotonicity). *Suppose $h : P \xrightarrow{\pi} Q$. Then $h\sigma : P\sigma \xrightarrow{\pi\sigma} Q\sigma$ for all σ that respect h and satisfy for all $x \in \mathrm{bn}(\pi)$, $y\sigma = x$ iff $x = y$.*

2.2 Four notions of bisimilarity

We augment early, late, open and quasi-open bisimilarity with histories. In each case, we first define a notion of bisimulation as a collection of relations indexed by the collection of histories. We write $\mathcal{H}$ for the collection of all histories, $\mathcal{H}^o$ for those consisting entirely of o-annotated names, and we similarly define $\mathcal{H}^i$. We let $\mathcal{H}^{i\text{-}o}$ denote the collection of histories in which every i-annotated name comes before all o-annotated names.

▶ **Definition 10** (Early bisimilarity). An *early bisimulation* is a family of symmetric relation $\{\mathcal{B}_e^h \mid h \in \mathcal{H}^o\}$ such that whenever $P\mathcal{B}_e^h Q$ we have:

- If $h : P \xrightarrow{\alpha} P'$ then $\exists Q'$ s.t. $h : Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{B}_e^h Q'$, where α is of the form τ or $\bar{x}y$.
- If $h : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh then $\exists Q'$ s.t. $h : Q \xrightarrow{\bar{x}(z)} Q'$ and $P'\mathcal{B}_e^{h\cdot z^o} Q'$.
- If $h : P \xrightarrow{x(z)} P'$ and z is fresh then for all $h' \supseteq_o h$ and $y \in h'$ there exists some Q' such that $h : Q \xrightarrow{x(z)} Q'$ and $P'\{{}^y\!/\!_z\}\mathcal{B}_e^{h'} Q'\{{}^y\!/\!_z\}$.

We write $\{\sim_e^h \mid h \in \mathcal{H}\}$ for the pointwise union of all early bisimulations and refer to $\sim_e^h$ as *early h-bisimilarity.* Two processes P and Q are called *early bisimilar* if they are early h-bisimilar for some $h \in \mathcal{H}^o$.

The third clause allows us to substitute z for any name y, including a name that does not appear in h (hence the need for the extension $h' \supseteq_o h$). The fact that we use only o-annotated histories in early (and late) bisimulation reflects the fact that names in these bisimulations cannot be instantiated, i.e., they are essentially constants. The definition of late bisimulation is similarly adapted from its original definition as follows.

▶ **Definition 11** (Late bisimilarity). A *late bisimulation* is a family $\{\mathcal{B}_\ell^h \mid h \in \mathcal{H}^o\}$ of symmetric relations indexed by a history consisting only of o-annotated names such that whenever $P\mathcal{B}_\ell^h Q$, we have:

- If $h : P \xrightarrow{\alpha} P'$ then $\exists Q'$ s.t. $h : Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{B}_\ell^h Q'$, where α is of the form τ or $\bar{x}y$;
- If $h : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh then $\exists Q'$ s.t. $h : Q \xrightarrow{\bar{x}(z)} Q'$ and $P'\mathcal{B}_\ell^{h\cdot z^o} Q'$;
- If $h : P \xrightarrow{x(z)} P'$ and z is fresh then $\exists Q'$ s.t. $h : Q \xrightarrow{x(z)} Q'$ and for all $y \in h'$ with $h' \supseteq_o h$ we have $P'\{{}^y\!/\!_z\}\mathcal{B}_\ell^{h'} Q'\{{}^y\!/\!_z\}$.

We write $\{\sim_\ell^h \mid h \in \mathcal{H}\}$ for the pointwise union of all late bisimulations and refer to $\sim_\ell^h$ as *late h-bisimilarity.* Two processes P and Q are called *late bisimilar* if they are late h-bisimlar for some $h \in \mathcal{H}^o$.

The notions of a late and early bisimilarity were originally defined in [14] without reference to a history. The original definition can be obtained from the above ones simply by omitting reference to the history, and in the third items letting y be an arbitrary name. We refer to this as *MPW late/early bisimulation*, and to the induced notion of bisimilarity as *MPW late/early bisimilarity.* The next proposition explains the connection between late/early bisimulations and MPW late/early bisimulations.

▶ **Proposition 12.** *Two processes are MPW late (resp. early) bisimilar if and only if they are late (resp. early) bisimilar.*

We now define analogues of open and quasi-open bisimulations [20, 22].

▶ **Definition 13** (Open bisimilarity). An *open bisimulation* is a history-indexed collection $\{\mathcal{B}_o^h \mid h \in \mathcal{H}\}$ of symmetric relations on processes such that whenever $P\mathcal{B}_e^h Q$:

- For all substitutions σ respecting h, we have $P\sigma\mathcal{B}_o^{h\sigma}Q\sigma$.
- If $h : P \xrightarrow{\alpha} P'$ then $\exists Q'$ s.t. $h : Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{B}_o^h Q'$, where α is of the form τ or $\bar{x}y$;
- If $h : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $\exists Q'$ s.t. $h : Q \xrightarrow{\bar{x}(z)} Q'$ and $P'\mathcal{B}_o^{h\cdot z^o}Q'$;
- If $h : P \xrightarrow{x(z)} P'$ and z is fresh, then $\exists Q'$ s.t. $h : Q \xrightarrow{x(z)} Q'$ and $P'\mathcal{B}_o^{h\cdot z^i}Q'$.

The pointwise union of all open bisimulations is denoted by $\{\sim_o^h\mid h \in \mathcal{H}\}$. We refer to $\sim_o^h$ as *open h-bisimilarity.* We write $P \sim_o^{h'} Q$ if there exists an open bisimulation $\{\mathcal{B}_o^h \mid h \in \mathcal{H}\}$ and a history h' with only i-annotated names such that $P\mathcal{B}_o^{h'}Q$ and $\mathrm{fn}(P, Q) \subseteq h'$. We call P and Q *open bisimilar.*

Augmenting open bisimulations to account for a history does not affect the resulting notion of bisimilarity compared to the original definition, as was shown in [26, Corollary 22].

Quasi-open bisimilarity was originally defined in [22] using the early transition semantics. We adapt the original definition into late transition semantics indexed by history.

▶ **Definition 14** (Quasi-open bisimilarity). A *quasi-open bisimulation* is a history-indexed family $\{\mathcal{B}_q^h \mid h \in \mathcal{H}^{i\text{-}o}\}$ of symmetric relations on processes such that whenever $P\mathcal{B}_q^h Q$:

- For all substitutions σ respecting h, we have $P\sigma\mathcal{B}_q^{h\sigma}Q\sigma$;
- If $h : P \xrightarrow{\alpha} P'$ then $\exists Q'$ s.t. $h : Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{B}_q^h Q'$, where $\alpha = \tau, \bar{x}y$;
- If $h : P \xrightarrow{\bar{x}(z)} P'$ and z is fresh, then $\exists Q'$ s.t. $h : Q \xrightarrow{\bar{x}(z)} Q'$ and $P'\mathcal{B}_q^{h\cdot z^o}Q'$;
- If $h : P \xrightarrow{x(z)} P'$ and z is fresh, then for all $h' \supseteq_i h$ and all $y \in h'$, $\exists Q'$ s.t. $h : Q \xrightarrow{x(z)} Q'$ and $P'\{{}^{y}\!/\!{}_{z}\}\mathcal{B}_q^{h'}Q'\{{}^{y}\!/\!{}_{z}\}$.

We write $\{\sim_q^h\mid h \in \mathcal{H}\}$ for the pointwise union of all quasi-open bisimulations and refer to $\sim_q^h$ as *quasi-open h-bisimilarity.* Two processes P and Q are called *quasi-open bisimilar* if there exists a history h with only i-annotated names such that $P \sim_q^h Q$ and $\mathrm{fn}(P, Q) \subseteq h$.

The first three conditions of an open and quasi-open bisimulation coincide. The last condition captures a subtle but important difference between open and quasi-open bisimulations: in quasi-open bisimulation, a bound output name must remain distinct from *all* other names produced during the bisimulation game, whereas in open bisimulation, the same bound output name only needs to be kept distinct from existing names in the history and future output names. This difference is captured technically by restricting the class of histories in quasi-open bisimulation to those where input names are always added to the front of output names, thereby preventing respectful substitutions from ever identifying output names with other (input/output) names.

Our definition relates to the original one in [22] as follows:

▶ **Proposition 15.** *Two processes are quasi-open bisimilar if and only if they are quasi-open bisimilar in the sense of [22].*

We use an example from [22] that distinguishes open and quasi-open bisimilarity to illustrate how to use histories.

▶ **Example 16.** Consider the processes

$$P = \nu u\bar{x}u.(x(z) + x(z).\tau + x(z).[z = u]\tau) \qquad Q = \nu u\bar{x}u.(x(z) + x(z).\tau)$$

We claim that P and Q are quasi-open bisimilar but not open bisimilar under the history $h = x^i$. After taking the transitions $\xrightarrow{\bar{x}(u)}\xrightarrow{x(z)}$ through the definition of open bisimilarity, we end up with the history $x^i \cdot u^o \cdot z^i$, while quasi-open bisimilarity yields history $y^i \cdot x^i \cdot u^o$ (if $y \notin \{x, u\}$) or $x^i \cdot u^o$ (if $y \in \{x, u\}$).

$$\begin{array}{lll}
P \models^h \texttt{tt} & & P \models^h x = x \qquad\qquad P \models^h \neg\varphi \quad \text{iff} \quad P \not\models^h \varphi. \\
P \models^h \varphi_1 \wedge \varphi_2 & \text{iff} & P \models^h \varphi_1 \text{ and } P \models^h \varphi_2 \\
P \models^h \Diamond\varphi & \text{iff} & \exists\sigma \text{ respecting } h, P\sigma \models^{h\sigma} \varphi\sigma \\
P \models^h \langle\alpha\rangle\varphi & \text{iff} & \exists Q \text{ s.t. } h : P \xrightarrow{\alpha} Q \text{ and } Q \models^h \varphi, \text{ where } \alpha = \tau, \bar{x}y \\
P \models^h \langle\bar{x}(z)\rangle\varphi & \text{iff} & \exists Q \text{ s.t. } h : P \xrightarrow{\bar{x}(z)} Q \text{ and } Q \models^{h\cdot z^o} \varphi \\
P \models^h \langle x(z)\rangle\varphi & \text{iff} & \exists Q \text{ s.t. } h : P \xrightarrow{x(z)} Q \text{ and } \exists y \in h'(h' \supseteq_o h), Q\{y/z\} \models^{h'} \varphi\{y/z\} \\
P \models^h \langle x(z)\rangle_\ell\varphi & \text{iff} & \exists Q \text{ s.t. } h : P \xrightarrow{x(z)} Q \text{ and } \forall y \in h'(h' \supseteq_o h), Q\{y/z\} \models^{h'} \varphi\{y/z\} \\
P \models^h \langle x(z)\rangle_e\varphi & \text{iff} & \forall y \in h'(h' \supseteq_o h), \exists Q \text{ s.t. } h : P \xrightarrow{x(z)} Q \text{ and } Q\{y/z\} \models^{h'} \varphi\{y/z\} \\
P \models^h \langle x(z)\rangle_o\varphi & \text{iff} & \exists Q, h : P \xrightarrow{x(z)} Q \text{ and } Q \models^{h\cdot z^i} \varphi \\
P \models^h \langle x(z)\rangle_q\varphi & \text{iff} & \forall y \in h'(h' \supseteq_i h), \exists Q \text{ s.t. } h : P \xrightarrow{x(z)} Q \text{ and } Q\{y/z\} \models^{h'} \varphi\{y/z\}
\end{array}$$

Figure 2 The semantics of logic $\mathbb{U}$. In each clause we require z to be fresh for h and σ, and that $\mathrm{fn}(P) \cup \mathrm{fn}(\varphi) \subseteq h$.

History $h = x^i \cdot u^o \cdot z^i$ indicates $x\sigma \neq u$ for all σ respecting it, while the substitution $\{u/z\}$ is allowed. After the transitions $\xrightarrow{\bar{x}(u)}\xrightarrow{x(z)}$, P can reach the state $[z = u]\tau$ and Q can reach either 0 or τ. Applying the substitution $\{u/z\}$ yields $[z = u]\tau \not\sim_o^h 0$, and applying $\{z/z\}$ gives $[z = u]\tau \not\sim_o^h \tau$. Therefore $P \not\sim_o^h Q$.

When considering quasi-open bisimilarity, if after the transitions $\xrightarrow{\bar{x}(u)}\xrightarrow{x(z)}$, $P' = 0$ or τ, then it is straightforward to show that P', Q' are quasi-open bisimilar. If $P' = [z = u]\tau$, we need to show that for all $h' \supseteq_i h$ and for all $y \in h'$, there exists a Q' such that $P'\{y/z\}$ is bisimilar to $Q'\{y/z\}$. Suppose $h' = h$ and $y \in \{x, u\}$. Then if $y = x$, we have that P' is bisimilar to $Q' = 0$, and if $y = u$ then P' is bisimilar to $Q' = \tau$. If $h' = y^i \cdot h$ and $y \notin \{x, u\}$ then P' is bisimilar to $Q' = 0$ because $y \neq u$. Therefore $P \sim_q^h Q$.

3 A universal logic

We define a universal logic $\mathbb{U}$ that characterises the four bisimilarities mentioned above.

▶ **Definition 17.** Let $\mathbb{U}$ be the language generated by the following grammar:

$$\begin{array}{rl}
\varphi ::= & \texttt{tt} \mid x = y \mid \varphi \wedge \varphi \mid \neg\varphi \mid \Diamond\varphi \mid \langle\alpha\rangle\varphi \mid \langle\bar{x}(z)\rangle\varphi \\
& \mid \langle x(z)\rangle\varphi \mid \langle x(z)\rangle_\ell\varphi \mid \langle x(z)\rangle_e\varphi \mid \langle x(z)\rangle_o\varphi \mid \langle x(z)\rangle_q\varphi
\end{array}$$

▶ **Definition 18.** Given a process P, a $\mathbb{U}$-formula φ and a history h, the satisfaction relation $P \models^h \varphi$ is defined in Figure 2.

▶ Remark. The modalities $\langle x(z)\rangle\varphi$, $\langle x(z)\rangle_\ell\varphi$ and $\langle x(z)\rangle_e\varphi$ correspond to the operators in MPW logic defined in [15]. The former is not used for our characterisations.

The dual logical propositional and modal connectives are defined as usual, via negation.

▶ **Definition 19** (Logical equivalence)**.** Two processes P and Q are *logically equivalent with respect to some* $\mathbb{S} \subseteq \mathbb{U}$ *and history* h, notation: $P \equiv_{\mathbb{S}}^h Q$, if $\mathrm{fn}(P) \cup \mathrm{fn}(Q) \subseteq h$ and for all $\varphi \in \mathbb{S}$ with $\mathrm{fn}(\varphi) \subseteq h$ we have $P \models^h \varphi$ iff $Q \models^h \varphi$.

We say that $\mathbb{S}$ characterises a history-indexed family $\{R^h \mid h \in \mathcal{H}\}$ of relations if: $P \equiv^h_{\mathbb{U}} Q$ iff PR^hQ. We define sublogics $\mathbb{E}$, $\mathbb{L}$, $\mathbb{Q}$ and $\mathbb{O}$ of $\mathbb{U}$ to characterise the late, early, open and quasi-open bisimilarity.

▶ **Definition 20** (Sublogics of $\mathbb{U}$)**.** The logics $\mathbb{E}$, $\mathbb{L}$, $\mathbb{Q}$ and $\mathbb{O}$ are the sublogics of $\mathbb{U}$ generated by the grammars with $\texttt{tt}, \neg, \wedge$ and the modalities specified in Table 1.

Table 1 The modalities defining the logics $\mathbb{E}, \mathbb{L}, \mathbb{Q}$ and $\mathbb{O}$.

Logic	Modalities	Characterises
$\mathbb{E}$	$\langle\alpha\rangle$, $\langle\bar{x}(z)\rangle$, $\langle x(z)\rangle_e$	early bisimilarity
$\mathbb{L}$	$\langle\alpha\rangle$, $\langle\bar{x}(z)\rangle$, $\langle x(z)\rangle_\ell$	late bisimilarity
$\mathbb{Q}$	$\Diamond$, $\langle\alpha\rangle$, $\langle\bar{x}(z)\rangle$, $\langle x(z)\rangle_q$	quasi-open bisimilarity
$\mathbb{O}$	$\Diamond$, $\langle\alpha\rangle$, $\langle\bar{x}(z)\rangle$, $\langle x(z)\rangle_o$	open bisimilarity

▶ **Theorem 21.** *The various notions of bisimilarity can be characterised as follows:*

1. $P \equiv^h_{\mathbb{E}} Q$ *iff* $P \sim^h_e Q$
2. $P \equiv^h_{\mathbb{L}} Q$ *iff* $P \sim^h_\ell Q$
3. $P \equiv^h_{\mathbb{Q}} Q$ *iff* $P \sim^h_q Q$
4. $P \equiv^h_{\mathbb{O}} Q$ *iff* $P \sim^h_o Q$

Proof. We postpone the completeness proofs (i.e. $P \equiv^h Q$ implies $P \sim^h Q$) to Section 4. The soundness proofs are straightforward. As an example, we demonstrate part of the soundness proof for open bisimilarity.

Suppose $P \sim^h_o Q$. Then by definition $\text{fn}(P, Q) \subseteq h$. Let φ be an $\mathbb{O}$-formula such that $\text{fn}(\varphi) \subseteq h$ and assume $P \models^h \varphi$. We prove that $Q \models^h \varphi$ by induction on the structure of φ. The propositional cases are routine. We showcase the modal cases for $\varphi = \Diamond\psi$ and $\varphi = \langle\bar{x}(z)\rangle\psi$. If $\varphi = \Diamond\psi$ then there exists a substitution σ respecting h such that $P\sigma \models^{h\sigma} \psi\sigma$. By definition of open bisimulation we have $P\sigma \sim^{h\sigma}_o Q\sigma$. Besides, $\text{fn}(\Diamond\psi) \subseteq h$ implies $\text{fn}(\psi) \subseteq h$, so the induction hypothesis gives $Q\sigma \models^{h\sigma} \psi\sigma$ and hence $Q \models^h \Diamond\psi$.

If $\varphi = \langle\bar{x}(z)\rangle\psi$ then there exists a P' such that $h : P \xrightarrow{\bar{x}(z)} P'$ and $P' \models^{h\cdot z^o} \psi$. By definition of $\models$ the name z is fresh for h. Therefore we can invoke the definition of open bisimulations to find a process Q' such that $h : Q \xrightarrow{\bar{x}(z)} Q'$ and $P' \sim^{h\cdot z^o}_o Q'$. Since $\text{fn}(\langle\bar{x}(z)\rangle\psi) \subseteq h$ we have $\text{fn}(\psi) \subseteq h \cdot z^o$, so we can use the induction hypothesis to derive $Q' \models^{h\cdot z^o} \psi$. Therefore $Q \models^h \langle\bar{x}(z)\rangle\psi$, as desired. ◀

We say a logic is sound and complete for a bisimilarity when: if two processes are bisimilar, then they satisfy the same set of formulas in the logic; and if two processes are not bisimilar, there exist some formulae in the logic that separate them, which we call the distinguishing formulae. A distinguishing formula holds for one process but not for the other, thus revealing a difference in their behavior. The distinguishing formulae can be used as efficiently checkable evidences to explain why two processes are not bisimilar.

The next examples illustrate how the modalities $\langle-\rangle_e$, $\langle-\rangle_\ell$, $\langle-\rangle_q$ and $\langle-\rangle_o$ can be used to recognise non-bisimilar processes.

▶ **Example 22** (Distinguishing processes that are not late bisimilar)**.** Consider the processes

$$P = x(z) + x(z).\tau + x(z).[z = u]\tau \qquad Q = x(z) + x(z).\tau$$

P and Q are early bisimilar but not late bisimilar (see e.g. [15, Section 2.3]). An induction on the structure of φ shows that we have $P \models^{h} \varphi$ iff $Q \models^{h} \varphi$ for all $h \in \mathcal{H}^{o}$ containing x^{o} and u^{o} and for all $\varphi \in \mathbb{E}$. However, if we bring $\langle x(z)\rangle_{\ell}$ into the picture then we can construct a formula which only holds at one of P and Q. For example, the formula

$$\varphi := \langle x(z)\rangle_{\ell}\big([\tau](z = u) \wedge ((z = u) \rightarrow \langle\tau\rangle\texttt{tt})\big)$$

is true at P but not at Q.

To show that $P \models^{x^{o}\cdot u^{o}} \varphi$ we need to find a process P' such that $x^{o} \cdot u^{o} : P \xrightarrow{x(z)} P'$ and for all $h' \supseteq_{o} x^{o} \cdot u^{o}$ and all $y \in h'$ we have $P' \models^{h'} \big([\tau](z = u) \wedge ((z = u) \rightarrow \langle\tau\rangle\texttt{tt})\big)\{{}^{y}\!/\!{}_{z}\}$. To this end, take $P' = [z = u]\tau$. Then we have $P'\{{}^{y}\!/\!{}_{z}\} \models^{h'} [\tau](z = u)\{{}^{y}\!/\!{}_{z}\}$, because if there is a τ-transition then we must have $(z = u)\{{}^{y}\!/\!{}_{z}\}$, and $P'\{{}^{y}\!/\!{}_{z}\} \models^{h'} ((z = u) \rightarrow \langle\tau\rangle\texttt{tt})\{{}^{y}\!/\!{}_{z}\}$ because if $(z = u)\{{}^{y}\!/\!{}_{z}\}$ is true then there must exists a τ-transition from P'.

To see that $Q \not\models^{x^{o}\cdot u^{o}} \varphi$, note that there are only two processes that Q can $x(z)$-transition to, namely $Q' = 0$ and $Q' = \tau$. In the first case we have $0\{{}^{u}\!/\!{}_{z}\} \not\models^{x^{o}\cdot u^{o}} [\tau](z = u) \wedge \big((z = u) \rightarrow \langle\tau\rangle\texttt{tt}\big)$ because the second conjoint is false, while in the second case we can take any $y \neq u$ to find $\tau \not\models^{x^{o}\cdot u^{o}\cdot y^{o}} [\tau](z = u) \wedge \big((z = u) \rightarrow \langle\tau\rangle\texttt{tt}\big)$ because the first conjoint is false.

▶ **Example 23.** Recall from Example 16 that the processes

$$P = \nu u\bar{x}u.(x(z) + x(z).\tau + x(z).[z = u]\tau), \qquad Q = \nu u\bar{x}u.(x(z) + x(z).\tau)$$

are quasi-open bisimilar but not open bisimilar. We construct a distinguishing formula using the modality $\langle x(z)\rangle_{o}$. First observe the difference between $[z = u]\tau$ and τ. The latter can always make a τ-transition while the former cannot do that without a suitable substitution. Therefore $[z = u]\tau \not\models^{x^{i}\cdot u^{o}\cdot z^{i}} \langle\tau\rangle\texttt{tt}$ while $\tau \models^{x^{i}\cdot u^{o}\cdot z^{i}} \langle\tau\rangle\texttt{tt}$. Similarly, the processes 0 and $[z = u]\tau$ can be distinguished by $[z = u]\tau \models^{x^{i}\cdot u^{o}\cdot z^{i}} \Diamond\langle\tau\rangle\texttt{tt}$ while $0 \not\models^{x^{i}\cdot u^{o}\cdot z^{i}} \Diamond\langle\tau\rangle\texttt{tt}$. Observe that both P and Q both can perform the transitions $\xrightarrow{\bar{x}(u)}\xrightarrow{x(z)}$ to arrive at states that are distinguishable by a formula. Thus, if we define $\varphi := \langle\bar{x}(u)\rangle\langle x(z)\rangle_{o}(\neg\langle\tau\rangle\texttt{tt} \wedge \Diamond\langle\tau\rangle\texttt{tt})$ then we have $P \models^{x^{i}} \varphi$ while $Q \not\models^{x_{i}} \varphi$.

4 Completeness for quasi-open and open bisimilarity

We now detail completeness for quasi-open bisimilarity. We first list here some useful lemmas that will be used in the main completeness proof. Most of these are straightforward to prove, except for Lemma 26, for which we outline a proof.

▶ **Definition 24.** Let h be a history and σ a substitution. Then we define

$$e(h,\sigma) := \bigwedge\{(x = y) \mid x, y \in h \text{ distinct}, \sigma(x) = \sigma(y)\} \wedge \bigwedge\{(x \neq y) \mid x, y \in h, \sigma(x) \neq \sigma(y)\}$$

This is a finite conjunction because h is finite.

▶ **Lemma 25.** *Let P be a process, h a history such that $\mathrm{fn}(P) \subseteq h$ and θ a renaming that respects h. Then for all formulas φ such that $\mathrm{fn}(\varphi) \subseteq h$ we have $P \models^{h} \varphi$ iff $P\theta \models^{h\theta} \varphi\theta$.*

▶ **Lemma 26.** *Let P be a process, h a history and σ a substitution that respects h. Then $P\sigma \models^{h\sigma} \varphi\sigma$ if and only if $P \models^{h} \Diamond(e(h,\sigma) \wedge \varphi)$.*

Proof. Suppose $P\sigma \models^{h\sigma} \varphi\sigma$. Since σ respects h, by definition of $e(h,\sigma)$ we have $P\sigma \models^{h\sigma} e(h,\sigma)\sigma$. Therefore $P\sigma \models^{h\sigma} (e(h,\sigma) \wedge \varphi)\sigma$, hence $P \models^{h} \Diamond(e(h,\sigma) \wedge \varphi)$.

For the conversely, suppose $P \models^h \Diamond(e(h,\sigma) \wedge \varphi)$. Then by definition of $\Diamond$, there exists substitution θ respecting h such that $P\theta \models^{h\theta} e(h,\sigma)\theta \wedge \varphi\theta$. Then we have $P\theta \models^{h\theta} e(h,\sigma)\theta$ and $P\theta \models^{h\theta} \varphi\theta$. By $P\theta \models^{h\theta} e(h,\sigma)\theta$ we have $x\sigma = y\sigma$ iff $x\theta = y\theta$ for all $x, y \in h$, and hence we can find a renaming θ' such that σ coincides with $\theta \cdot \theta'$ on h (i.e. $z\sigma = (z\theta)\theta'$ for all $z \in h$). Moreover we may assume that θ' respects h. Then we can use Lemma 25 and the assumption $P\theta \models^{h\theta} \varphi\theta$ to find that $P\sigma \models^{h\sigma} \varphi\sigma$. ◀

▶ **Lemma 27.**

1. *If $h\sigma : P\sigma \xrightarrow{\pi} P'$, then there exists an action π' such that $\pi = \pi'\sigma$.*
2. *If $P\sigma \models^{h\sigma} \varphi$ then there exists a formula φ' using the same connectives as φ s.t. $\varphi'\sigma = \varphi$.*

▶ **Lemma 28** (image finiteness). *For any process P and action π there are finitely many P_i, up to renaming of* $\mathrm{bn}(\pi)$*, such that $P \xrightarrow{\pi} P_i$.*

To prove the completeness is equivalent to prove that if two processes are not bisimilar then there must be some distinguishing formulae that can be satisfied by one of the processes but by not the other. The proof will provide a strategy on constructing distinguishing formulae for any processes that are not quasi-open bisimilar. On the basis of the image finiteness, we are able to define distinguishability, which is a negation of bisimilarity. Note that the subscript of $\sim_0$ below is the number zero instead of the letter o for open bisimilarity.

▶ **Definition 29** (distinguishability). Let $\not\sim_0^h$ be the smallest symmetric relation satisfying $P \not\sim_0^h Q$ whenever there exists a substitution σ respecting h and an action π such that

- There exists a P' such that $h\sigma : P\sigma \xrightarrow{\pi} P'$ but no Q' satisfying $h\sigma : Q\sigma \xrightarrow{\pi} Q'$.

If π is of the shape $\bar{x}(z)$ or $x(z)$ then we assume that z is fresh for h and σ.

We inductively define $\not\sim_{n+1}^h$ as the smallest symmetric relation containing $\not\sim_n^h$ such that $P \not\sim_{n+1}^h Q$ holds if there exists a σ respecting h and a process P' such that either

- $h\sigma : P\sigma \xrightarrow{\alpha} P'$ and for all Q' such that $h\sigma : Q\sigma \xrightarrow{\alpha} Q'$ we have $P' \not\sim_n^{h\sigma} Q'$, where α is of the form τ or $\bar{x}y$; or
- $h\sigma : P\sigma \xrightarrow{\bar{x}(z)} P'$ for some $z \notin h$ and for all Q' such that $h\sigma : Q\sigma \xrightarrow{\bar{x}(z)} Q'$ we have $P' \not\sim_n^{h\sigma \cdot z^o} Q'$; or
- $h\sigma : P\sigma \xrightarrow{x(z)} P'$ for some $z \notin h$ and there exists some $h' \supseteq_i h$ and $y \in h'$ such that for all Q' with $h\sigma : Q\sigma \xrightarrow{x(z)} Q'$ we have $P'\{y/z\} \not\sim_n^{h'\sigma} Q'\{y/z\}$.

Again, if π is of the shape $\bar{x}(z)$ or $x(z)$ then we assume that z is fresh for h and σ.

▶ **Lemma 30.** *Let P and Q be processes and $h \in \mathcal{H}^{i\text{-}o}$ a history such that* $\mathrm{fn}(P,Q) \subseteq h$*. Then $P \not\sim_q^h Q$ if and only if there exists some n such that $P \not\sim_n^h Q$.*

Theorem 21(3), that is, completeness of quasi-open bisimilarity with respect to $\mathbb{Q}$, follows immediately from the next lemma.

▶ **Lemma 31.** *If $P \not\sim_q^h Q$, then there exists $\varphi \in \mathbb{Q}$ such that $P \models_{\mathbb{Q}}^h \varphi$ and $Q \not\models_{\mathbb{Q}}^h \varphi$.*

Proof. If $P \not\sim_q^h Q$ then there exists some n such that $P \not\sim_n^h Q$ by Lemma 30. We now construct a distinguishing formula using induction on n. The base case is straightforward. We show here a non-trivial inductive case.

Suppose $P \not\sim_{n+1}^h Q$. Without loss of generality assume that there exists a substitution σ respecting h and a process P such that on of the three cases from Definition 29 holds.

Case 1: $P\sigma \xrightarrow{\alpha} P'$ and for all Q' that satisfy $h\sigma : Q\sigma \xrightarrow{\alpha} Q'$ we have $P' \not\sim_n^{h\sigma} Q'$, where α is of the form τ or $\bar{x}y$. The case is similar to case 2 below.

Case 2: $P\sigma \xrightarrow{\bar{x}(z)} P'$ for some $z \notin h$ and for all Q' that satisfy $h\sigma : Q\sigma \xrightarrow{\bar{x}(z)} Q'$ we have $P' \not\sim_n^{h\sigma \cdot z^o} Q'$. By Lemma 28 there are finitely many such Q' (up to renaming of bound names in π) such that $Q\sigma \xrightarrow{\bar{x}(z)} Q'$, so we can enumerate them $Q'_1, \dots, Q'_m$. Since $P' \not\sim_n^{h\sigma \cdot z^o} Q'_i$, the induction hypothesis yields a formula φ_i such that $P' \models^{h\sigma \cdot z^o} \varphi_i$ while $Q'_i \not\models^{h\sigma \cdot z^o} \varphi_i$, for all $i \in \{1, \dots, m\}$. We claim that

$$P \models^h \Diamond(e(h,\sigma) \wedge \langle \bar{x}(z) \rangle \bigwedge_i \varphi'_i) \quad \text{and} \quad Q \not\models^h \Diamond(e(h,\sigma) \wedge \langle \bar{x}(z) \rangle \bigwedge_i \varphi'_i)$$

where $\varphi'_i\sigma = \varphi_i$. By Lemma 27, we have $P\sigma \xrightarrow{\overline{x\sigma}(z)} P'$, and $P' \models^{h\sigma \cdot z^o} \varphi'_i\sigma$, where $\varphi'_i\sigma = \varphi_i$. Then $P\sigma \models^{h\sigma} \langle \overline{x\sigma}(z) \rangle \varphi'_i\sigma$. Since the above holds for all i, $P\sigma \models^{h\sigma} \langle \overline{x\sigma}(z) \rangle \bigwedge_i \varphi'_i\sigma$. Lemma 26 then gives we have $P \models^h \Diamond(e(h,\sigma) \wedge \langle \bar{x}(z) \rangle \bigwedge_i \varphi'_i)$.

For the latter, assume otherwise that $Q \models^h \Diamond(e(h,\sigma) \wedge \langle \bar{x}(z) \rangle \bigwedge_i \varphi'_i)$, then by Lemma 26, $Q\sigma \models^{h\sigma} \langle \overline{x\sigma}(z) \rangle \bigwedge_i \varphi'_i\sigma)$, then there exists Q'_i such that $Q\sigma \xrightarrow{\overline{x\sigma}(z)} Q'_i$ and $Q'_i \models^{h\sigma \cdot z^o} \bigwedge_i \varphi_i$, contradicting the condition that no such Q'_i exists.

Case 3: $h\sigma : P\sigma \xrightarrow{x(z)} P'$ for some $z \notin h\sigma$ and there exists a $h' \supseteq_i h\sigma$ and a $y_0 \in h'$ such that for all Q' that satisfy $h\sigma : Q\sigma \xrightarrow{x(z)} Q'$ we have $P'\{y_0/z\} \not\sim_n^{h'} Q'\{y_0/z\}$. We may assume that $h' = h\sigma$ if $y_0 \in h\sigma$ and $h' = y_0^i \cdot h\sigma$ otherwise. By Lemma 28 there are finitely many such Q' (up to renaming of bound names in π), so we can enumerate them $Q'_1, \dots, Q'_m$. Since $P'\{y_0/z\} \not\sim_n^{h'\sigma} Q'_i\{y_0/z\}$ by the induction hypothesis we can find a formula φ_i such that

$$P'\{y_0/z\} \models^{h'} \varphi_i \quad \text{while} \quad Q'_i\{y_0/z\} \not\models^{h'} \varphi_i,$$

for all $i \in \{1, \dots, m\}$. Let $\varphi'_i = \varphi_i\{z/y_0\}$ (so obviously $\varphi_i = \varphi'_i\{y_0/z\}$).

Setting $\varphi := \varphi'_1 \wedge \dots \wedge \varphi'_m$, we have

$$P'\{y_0/z\} \models^{h'} \varphi\{y_0/z\} \quad \text{while} \quad Q'_i\{y_0/z\} \not\models^{h'} \varphi\{y_0/z\}, \tag{1}$$

for all $i \in \{1, \dots, m\}$. We now consider two subcases:

Case 3A: $y_0 \in h\sigma$. We now claim that

$$P\sigma \models^{h\sigma} \langle x(z) \rangle_q ((z = y_0) \to \varphi) \quad \text{but} \quad Q\sigma \not\models^{h\sigma} \langle x(z) \rangle_q ((z = y_0) \to \varphi). \tag{2}$$

For the former, let h'' be any history such that $h'' \supseteq_i h\sigma$ and let $y \in h''$. We need to find some P'' such that $h\sigma : P \xrightarrow{x(z)} P''$ and $P'' \models^{h''} ((z = y_0) \to \varphi)\{y/z\}$. Again, we may assume that $h'' = h$ if $y \in h$ and $h'' = y^i \cdot h$ otherwise. Take $P'' = P'$. Then we know that $h\sigma : P\sigma \xrightarrow{x(z)} P'$, so we only need to show that $P' \models^{h''} (z = y_0)\{y/z\} \to \varphi\{y/z\}$. If $y \neq y_0$ then $(z = y_0)\{y/z\}$ is false so the implication is true. If $y = y_0$ then $h'' = h'$ and $\varphi\{y/z\} = \varphi\{y_0/z\}$ so that (1) implies $P'\{y/z\} \models^{h''} \varphi\{y/z\}$, as desired.

Now for the latter, consider history h' and $y_0 \in h'$. Then (clearly) for all Q'_i we have $Q'_i \models^{h'} (z = y_0)\{y_0/z\}$. But $Q'_i \not\models^{h'} \varphi\{y_0/z\}$ by (1), so $Q'_i \not\models^{h'} ((z = y_0) \to \varphi)\{y_0/z\}$. Since the Q'_i range over the $x(z)$-successors of Q (up to renaming of bound names in π), this implies $Q \not\models^{h\sigma} \langle x(z) \rangle_q ((y_0 = z) \to \varphi)$.

Now by Lemma 27 we can find a ψ such that $\psi\sigma = \langle x(z) \rangle_q ((x = y_0) \to \varphi)$, so $P\sigma \models^{h\sigma} \psi\sigma$ and $Q\sigma \not\models^{h\sigma} \psi\sigma$. Lemma 26 then a distinguishing formula which is true at P but false at Q.

Case 3B: $y_0 \notin h\sigma$. Let φ' be $\varphi\{z/y_0\}$. We now claim that

$$P\sigma \models^{h\sigma} \langle x(z) \rangle_q (z \notin h\sigma \to \varphi) \quad \text{but} \quad Q\sigma \not\models^{h\sigma} \langle x(z) \rangle_q (z \notin h\sigma \to \varphi). \tag{3}$$

where $z \notin h\sigma$ refers to $\bigwedge\{z \neq w \mid w \in h\sigma\}$. This case follows a similar reasoning as in 3A, with the inequality guard $(z \notin h\sigma)$ replacing the role of $(z = y_0)$ in 3A.

The symmetric cases, with the role of P and Q reversed, can be obtained by taking the negated distinguishing formula constructed above. ◀

The proofs for early and late bisimilarity resemble the one above. For open bisimilarity, the definition for $P \not\sim^h_{n+1} Q$ (Definition 29) differs in the third clause, which is changed to

$$\exists P', h\sigma : P\sigma \xrightarrow{x\sigma(z)} P' \text{ and } \forall Q_i \text{ such that } h\sigma : Q\sigma \xrightarrow{x\sigma(z)} Q_i, P' \not\sim^{h\sigma \cdot z^i}_n Q_i$$

as the name z^i is added at a different location in history compared to quasi-open bisimilarity. The remainder of the proof proceeds along the line of the completeness proof for quasi-open bisimulation, but with all three inductive steps resembling cases 1 and 2. In fact, the proof can also be derived from the connection with [5] outlined in Section 6 below.

5 Handling mismatch

So far our language does not include the mismatch prefix $[x \neq y]$, with the interpretation that $[x \neq y]P$ can proceed as P only if x and y are not equal. Adding mismatch is problematic because doing so naively may invalidate Lemma 9 (monotonicity), which requires that "any name-substitution to a process does not diminish its capabilities for action" [23, Chapter 1.1]. In the context of open and quasi-open bisimulation, since names in a process may be subjected to instantiations, the operational semantics for mismatch need to account for all possible instantiations. This is easy to accommodate when the semantics is augmented with histories. The following rule for mismatch is an adaptation of a similar rule in [10]:

$$\frac{h : P \xrightarrow{\pi} Q \quad h \models x \neq y}{h : [x \neq y]P \xrightarrow{\pi} Q} \ (\textsc{Mismatch})$$

where $h \models x \neq y$ iff $x\sigma \neq y\sigma$ for all substitutions σ respecting h.

The monotonicity lemma (Lemma 9) still holds even in the presence of the Mismatch rule. We sketch here a proof for the inductive step. Let σ be any substitution which is respectful with respect to h. First observe that Lemma 7 implies that $h\sigma \models x\sigma \neq y\sigma$. (Indeed, if $h\sigma \not\models x\sigma \neq y\sigma$ then there exists a respectful substitution θ such that $(x\sigma)\theta = (y\sigma)\theta$, but then $\sigma\theta$ respects h and identifies x and y, a contradiction.) By induction hypothesis we have $h\sigma : P\sigma \xrightarrow{\pi\sigma} Q\sigma$. Thus we can use the mismatch rule to find $h\sigma : ([x \neq y]P)\sigma \xrightarrow{\pi\sigma} Q\sigma$.

Monotonicity aside, there is still a problem with mismatch: closure of (quasi-)open bisimilarity under restriction no longer holds. For example, under current definitions, the process $[x \neq y]\tau$ under the history $h = x^i \cdot y^i$ is open and quasi-open bisimilar with 0, since there is a respectful substitution that could invalidate $x \neq y$ (i.e., $\{x/y\}$) so that the τ-transition is not possible from $[x \neq y]\tau$. But neither open-bisimilarity nor quasi-open bisimilarity holds for $\nu y.[x \neq y]\tau$ and 0 under the history $h' = x^i$, since $[x \neq y]$ is always true when y is restricted. To solve this problem, we need to close the (quasi-)open bisimilarity with *rigidisation of names*, i.e., turning an i-annotated name into o-annotated. We extend our logic $\mathbb{U}$ with another accessibility relation that is induced by rigidisation, in addition to the accessibility relation induced by respectful substitution.

We first define the rigidisation relations.

▶ **Definition 32** (Rigidisation relations)**.** The relations $\subseteq_{ro}$ and $\subseteq_{rq}$ are the smallest relations on histories such that:

- $h \subseteq_{ro} h'$ iff $h = h_1 \cdot x^i \cdot h_2$ and $h' = h_1 \cdot x^o \cdot h_2$.
- $h \subseteq_{rq} h'$ iff $h = h_1 \cdot x^i \cdot h_2$ and $h' = h_1 \cdot h_2 \cdot x^o$.
- $h \subseteq_{ro} h \subseteq_{rq} h$.
- Both $\subseteq_{ro}$ and $\subseteq_{rq}$ are transitively closed.

We extend our logic $\mathbb{U}$ with new modal operators $\Diamond_{ro}$ and $\Diamond_{rq}$ for rigidisation of names in open and quasi-open bisimilarities respectively. The semantics are defined as follows.

$$
\begin{array}{lll}
P \models^h \Diamond_{ro}\varphi & \text{iff} & \exists h' \supseteq_{ro} h, P \models^{h'} \varphi \\
P \models^h \Diamond_{rq}\varphi & \text{iff} & \exists h' \supseteq_{rq} h, P \models^{h'} \varphi
\end{array}
$$

Accordingly, the definitions of open and quasi-open bisimilarity also need to be extended. For open bisimilarity, we add an additional clause to Definition 13:

For any $h' \supseteq_{ro} h$, we have $P\mathcal{B}_o^{h'}Q$.

For quasi-open bisimilarity, we add an additional clause to Definition 14:

For any $h' \supseteq_{rq} h$, we have $P\mathcal{B}_q^{h'}Q$.

We conjecture that the extensions of bisimilarities we defined here are the same as the definitions given by [10], and that they are characterised by our extended logic $\mathbb{U}$.

6 Related work

The idea of accounting of history of names in process transitions has been considered in other settings, notably in the automata theoretic model called *history-dependent automata* [16, 17]. In the case of bisimilarity relations, indexing the relations with a context more general than distinctions has also been considered in work on *environmental bisimulation* [21], and bisimulations for cryptographic calculi,e.g., [3, 6, 24]. In particular, our notion of histories is a special instance of that used in [24]. None of these works consider specifically the problem of characterising bisimulations via logic. We discuss next two other closest related works.

Relations between sublogic $\mathbb{O}$ and the intuitionistic modal logic $\mathbb{OM}$

In [5], open bisimilarity is characterised $\mathbb{OM}$, which extends intuitionistic logic with modalities of the form $\langle\pi\rangle\varphi$ and $[\pi]\varphi$ where π is of the form τ, $\bar{x}y$, $\bar{x}(z)$ or $x(z)$. The diamonds are interpreted with respect to a process and a history as in Definition 18 above, with $\langle x(z)\rangle$ being interpreted as $\langle x(z)\rangle_o$. The box operators are interpreted as the duals of the diamonds, with the additional condition that they be closed under respectful substitutions. For example,

$$P \models^h_{\mathbb{OM}} [\bar{x}(z)]\varphi \quad \text{iff} \quad \forall\sigma \text{ respecting } h, \forall Q,\ P\sigma \xrightarrow{\overline{x\sigma}z} Q \text{ implies } Q \models^{h\sigma}_{\mathbb{OM}} \varphi\sigma.$$

The logic $\mathbb{OM}$ can faithfully be embedded in $\mathbb{O}$ via a variation of the Gödel-McKinsey-Tarski translation of intuitionistic logic into the modal logic **S4** (see e.g. [7, §3.9]).

▶ **Definition 33.** Define the translation $t : \mathbb{OM} \to \mathbb{O}$ on propositional connectives as the Gödel-McKinsey-Tarski translation:

$$
\begin{array}{lll}
t(\texttt{tt}) = \texttt{tt} & t(\varphi_1 \land \varphi_2) = t(\varphi_1) \land t(\varphi_2) & t(x = y) = (x = y) \\
t(\texttt{ff}) = \texttt{ff} & t(\varphi_1 \lor \varphi_2) = t(\varphi_1) \lor t(\varphi_2) & t(\varphi_1 \supset \varphi_2) = \Box(t(\varphi_1) \to t(\varphi_2))
\end{array}
$$

This is extended to modalities as follows:

$$
\begin{array}{lll}
t(\langle\pi\rangle\varphi) = \langle\pi\rangle t(\varphi) & t([\pi]\varphi) = \Box([\pi]t(\varphi)) & \text{for } \pi = \tau, \bar{x}y, \bar{x}(z) \\
t(\langle\pi\rangle\varphi) = \langle\pi\rangle_o t(\varphi) & t([\pi]\varphi) = \Box([\pi]_o t(\varphi)) & \text{for } \pi = x(z)
\end{array}
$$

The correctness of the translation relies of the following lemma.

▶ **Lemma 34.** *For all P and all $\varphi \in \mathbb{OM}$ we have $P \models^h_{\mathbb{OM}} \varphi$ iff $P \models^h t(\varphi)$.*

Completeness for open bisimilarity now follows from [5, Theorem 3.3]: If $P \not\sim^h_o Q$ then there exists a formula $\varphi \in \mathbb{OM}$ such that $P \models^h_{\mathbb{OM}} \varphi$ while $Q \not\models^h_{\mathbb{OM}} \varphi$ or vice versa. Lemma 34 implies that $t(\varphi) \in \mathbb{O}$ satisfies $P \models^h t(\varphi)$ while $Q \not\models^h t(\varphi)$ (or vice versa), so that $P \not\equiv^h_o Q$.

Modal logics for nominal transition systems

In [18], Parrow et. al., defines a general framework for defining transition systems, called nominal transition systems, that subsumes most of name-passing calculi, including the π-calculus. They then define a general modal logic that characterises bisimilarity relations defined on nominal transition systems. They show several examples of how their framework can be instantiated to provide logical characterisations of bisimilarity relations; these include the π-calculus (without mismatch) and early, late and open bisimilarity. Of particular interests in the context of the current paper is the way in which they capture the notion of respectful substitutions, which is formalised as a notion of effects. In the modal logic for open bisimilarity, their logic considers an operator @ that applies an effect to the state (i.e., a process) of their semantic judgment, e.g., $P \models f@\varphi$ iff $f(P) \models \varphi$. This resembles our operator $\diamond$, however there are a couple of crucial differences: we apply the substitution (effect f in this example) to both the state and the modal formula, and our logic contains the equality predicate whereas their logic (for this particular example involving the π-calculus) does not allow equality (or any state predicates). The equality predicate becomes quite crucial when mismatch is present and at this stage, it is not clear whether quasi-open bisimilarity with mismatch can be similarly defined in the framework of nominal transition systems.

7 Conclusion

In this paper we considered early, late, open and quasi-open bisimilarity for the finite fragment of the π-calculus extended with the mismatch operator. We provided a unified presentation of each of these notions in the late transition semantics, using the notion of a history to capture the name context of a process. We then defined a unifying modal logic, and identified four fragments charaterising the four notions of bisimilarity. That is, for each type of bisimilarity we gave a sublogic of the unifying logic such that two processes are bisimilar if and only if they satisfy precisely the same formulas in the fragment.

As a consequence of the fact that our unifying logic is classical, we obtain a simple construction of distinguishing formulas for non-bisimilar processes in the context of open and quasi-open bisimilarity, compared to [5, 10].

An interesting direction for further research is to investigate to what extend our unifying logic can be used for extensions of our fragment of the π-calculus to include e.g. replication or recursion, or to cryptographic calculi such as the spi-calculus [2] or the applied π-calculus [1]. Some work in this direction can be found in [8, 12, 19, 25, 11].

References

1 Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018. doi:10.1145/3127586.

2 Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In Richard Graveman, Philippe A. Janson, Clifford Neuman, and Li Gong, editors, *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997*, pages 36–47. ACM, 1997. doi:10.1145/266420.266432.

3 Martín Abadi and Andrew D. Gordon. A bisimulation method for cryptographic protocols. *Nord. J. Comput.*, 5(4):267, 1998.

4 Ki Yung Ahn, Ross Horne, and Alwen Tiu. A characterisation of open bisimilarity using an intuitionistic modal logic. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 7:1–7:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.CONCUR.2017.7`.

5 Ki Yung Ahn, Ross Horne, and Alwen Tiu. A characterisation of open bisimilarity using an intuitionistic modal logic. *Logical Methods in Computer Science*, 17, 2021.

6 Johannes Borgström and Uwe Nestmann. On bisimulations for the spi calculus. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2002. `doi:10.1007/3-540-45719-4_20`.

7 A. Chagrov and M. Zakharyaschev. *Modal Logic.* Oxford University Press, Oxford, 1997.

8 Ulrik Frendrup, Hans Hüttel, and Jesper Nyholm Jensen. Modal logics for cryptographic processes. In Uwe Nestmann and Prakash Panangaden, editors, *9th International Workshop on Expressiveness in Concurrency, EXPRESS 2002, Satellite Workshop from CONCUR 2002, Brno, Czech Republic, August 19, 2002*, volume 68 of *Electronic Notes in Theoretical Computer Science*, pages 124–141. Elsevier, 2002. `doi:10.1016/S1571-0661(05)80368-8`.

9 Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985. `doi:10.1145/2455.2460`.

10 Ross Horne, Ki Yung Ahn, Shang-Wei Lin, and Alwen Tiu. Quasi-open bisimilarity with mismatch is intuitionistic. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 26–35. ACM, 2018. `doi:10.1145/3209108.3209125`.

11 Ross Horne and Sjouke Mauw. Discovering epassport vulnerabilities using bisimilarity. *Log. Methods Comput. Sci.*, 17(2):24, 2021. `doi:10.23638/LMCS-17(2:24)2021`.

12 Hans Hüttel and Michael D. Pedersen. A logical characterisation of static equivalence. In Marcelo Fiore, editor, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2007, New Orleans, LA, USA, April 11-14, 2007*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 139–157. Elsevier, 2007. `doi:10.1016/j.entcs.2007.02.032`.

13 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. `doi:10.1007/3-540-10235-3`.

14 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information & Computation*, 100:41–77, 1992. `doi:10.1016/0890-5401(92)90009-5`.

15 Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theor. Comput. Sci.*, 114(1):149–171, 1993. `doi:10.1016/0304-3975(93)90156-N`.

16 Ugo Montanari and Marco Pistore. An introduction to history dependent automata. In Andrew D. Gordon, Andrew M. Pitts, and Carolyn L. Talcott, editors, *Second Workshop on Higher-Order Operational Techniques in Semantics, HOOTS 1997, Stanford, CA, USA, December 8-12, 1997*, volume 10 of *Electronic Notes in Theoretical Computer Science*, pages 170–188. Elsevier, 1997. `doi:10.1016/S1571-0661(05)80696-6`.

17 Ugo Montanari and Marco Pistore. History dependent automata. Technical report, Università di Pisa, 1998.

18 Joachim Parrow, Johannes Borgström, Lars-Henrik Eriksson, Ramunas Gutkovas, and Tjark Weber. Modal logics for nominal transition systems. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: `https://lmcs.episciences.org/7137`.

19 Michael David Pedersen. Logics for the applied pi calculus. *BRICS Report Series*, 13(19), December 2006. `doi:10.7146/brics.v13i19.21923`.

20 Davide Sangiorgi. A theory of bisimulation for the pi-calculus. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 1993. doi:10.1007/3-540-57208-2_10.

21 Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011. doi:10.1145/1889997.1890002.

22 Davide Sangiorgi and David Walker. On barbed equivalences in pi-calculus. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 – Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2001. doi:10.1007/3-540-44685-0_20.

23 Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

24 Alwen Tiu. A trace based bisimulation for the spi calculus: An extended abstract. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2007. doi:10.1007/978-3-540-76637-7_25.

25 Alwen Tiu and Jeremy E. Dawson. Automating open bisimulation checking for the spi calculus. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 307–321. IEEE Computer Society, 2010. doi:10.1109/CSF.2010.28.

26 Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the pi-calculus. *ACM Trans. Comput. Log.*, 11(2):13:1–13:35, 2010. doi:10.1145/1656242.1656248.

Deciding What Is Good-For-MDPs

Sven Schewe
University of Liverpool, UK

Qiyi Tang
University of Liverpool, UK

Tansholpan Zhanabekova
University of Liverpool, UK

Abstract

Nondeterministic good-for-MDPs (GFM) automata are for MDP model checking and reinforcement learning what good-for-games automata are for reactive synthesis: a more compact alternative to deterministic automata that displays nondeterminism, but only so much that it can be resolved locally, such that a syntactic product can be analysed. GFM has recently been introduced as a property for reinforcement learning, where the simpler Büchi acceptance conditions it allows to use is key. However, while there are classic and novel techniques to obtain automata that are GFM, there has not been a decision procedure for checking whether or not an automaton is GFM. We show that GFM-ness is decidable and provide an EXPTIME decision procedure as well as a PSPACE-hardness proof.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Mathematics of computing → Markov processes

Keywords and phrases Büchi automata, Markov Decision Processes, Omega-regular objectives, Reinforcement learning

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.35

Related Version *Full Version*: https://arxiv.org/abs/2202.07629 [16]

Funding This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 956123 (FOCETA).

Acknowledgements We thank the anonymous reviewers of this paper for their constructive feedback. We thank an anonymous reviewer for raising the excellent question (to the original version) of whether or not QGFM and GFM are different. They proved not to be. Without their clever question, we would not have considered this question, and thus not strengthened this paper accordingly.

1 Introduction

Omega-automata [20, 12] are formal acceptors of ω-regular properties, which often result from translating a formula from temporal logics like LTL [14], as a specification for desired model properties in quantitative model checking and strategy synthesis [3], and reinforcement learning [19].

Especially for reinforcement learning, having a simple Büchi acceptance mechanism has proven to be a breakthrough [8], which led to the definition of the "good-for-MDPs" property in [9]. Just like for good-for-games automata in strategy synthesis for strategic games [10], there is a certain degree of nondeterminism allowed when using a nondeterministic automaton on the syntactic product with an MDP to learn how to control it, or to apply quantitative model checking. Moreover, the degree of freedom available to control MDPs is higher than the degree of freedom for controlling games. In particular, this always allows for using nondeterministic automata with a Büchi acceptance condition, both when using the classically used suitable limit deterministic automata [21, 6, 7, 17, 8] and for alternative GFM automata like the slim automata from [9].

This raises the question of whether or not an automaton is good-for-MDPs. While [9] has introduced the concept, there is not yet a decision procedure for checking the GFM-ness of an automaton, let alone for the complexity of this test.

We will start by showing that the problem of deciding GFM-ness is PSPACE-hard by a reduction from the NFA universality problem [18]. We then define the auxiliary concept of *qualitative GFM*, QGFM, which relaxes the requirements for GFM to qualitative properties, and develop an automata based EXPTIME decision procedure for QGFM. This decision procedure is constructive in that it can provide a counter-example for QGFM-ness when such a counter-example exists. We then use it to provide a decision procedure for GFM-ness that uses QGFM queries for all states of the candidate automaton. Finally, we show that the resulting criterion for GFM-ness is also a necessary criterion for QGFM-ness, which leads to a collapse of the two concepts. This entails that the EXPTIME decision procedure we developed to test QGFM-ness can be used to decide GFM-ness, while our PSPACE-hardness proofs extend to QGFM-ness.

2 Preliminaries

We write $\mathbb{N}$ for the set of nonnegative integers. Let S be a finite set. We denote by $\mathrm{Distr}(S)$ the set of probability distributions on S. For a distribution $\mu \in \mathrm{Distr}(S)$ we write $\mathsf{support}(\mu) = \{s \in S \mid \mu(s) > 0\}$ for its support. The cardinal of S is denoted $|S|$. We use Σ to denote a finite alphabet. We denote by Σ^* the set of finite words over Σ and Σ^ω the set of ω-words over Σ. We use the standard notions of prefix and suffix of a word. By $w\alpha$ we denote the concatenation of a finite word w and an ω-word α. If $L \subseteq \Sigma^\omega$ and $w \in \Sigma^*$, the residual language (left quotient of L by w), denoted by $w^{-1}L$ is defined as $\{\alpha \in \Sigma^\omega \mid w\alpha \in L\}$.

2.1 Automata

A nondeterministic Büchi **word automaton** (NBW) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and $F \subseteq Q$ is the set of final (or accepting) states. An NBW is *complete* if $\delta(q, \sigma) \neq \emptyset$ for all $q \in Q$ and $\sigma \in \Sigma$. Unless otherwise mentioned, we consider complete NBWs in this paper. A run r of $\mathcal{A}$ on $w \in \Sigma^\omega$ is an ω-word $q_0, w_0, q_1, w_1, \ldots \in (Q \times \Sigma)^\omega$ such that $q_i \in \delta(q_{i-1}, w_{i-1})$ for all $i > 0$. An NBW $\mathcal{A}$ accepts exactly those runs, in which at least one of the infinitely often occurring states is in F. A word in Σ^ω is accepted by the automaton if it has an accepting run, and the language of an automaton, denoted $\mathcal{L}(\mathcal{A})$, is the set of accepted words in Σ^ω. An example of an NBW is given in Figure 1.

Let $C \subset \mathbb{N}$ be a finite set of colours. A nondeterministic parity **word automaton** (NPW) is a tuple $P = (\Sigma, Q, q_0, \delta, \pi)$, where Σ, Q, q_0 and δ have the same definitions as for NBW, and $\pi : Q \to C$ is the priority (colouring) function that maps each state to a priority (colour). A run is accepting if and only if the highest priority (colour) occurring infinitely often in the infinite sequence is even. Similar to NBW, a word in Σ^ω is accepted by an NPW if it has an accepting run, and the language of the NPW P, denoted $\mathcal{L}(P)$, is the set of accepted words in Σ^ω. An NBW is a special case of an NPW where $\pi(q) = 2$ for $q \in F$ and $\pi(q) = 1$ otherwise with $C = \{1, 2\}$.

A nondeterministic word automaton is *deterministic* if the transition function δ maps each state and letter pair to a singleton set, a set consisting of a single state.

A nondeterministic automaton is called *good-for-games (GFG)* if it only relies on a limited form of nondeterminism: GFG automata can make their decision of how to resolve their nondeterministic choices on the history at any point of a run rather than using the knowledge

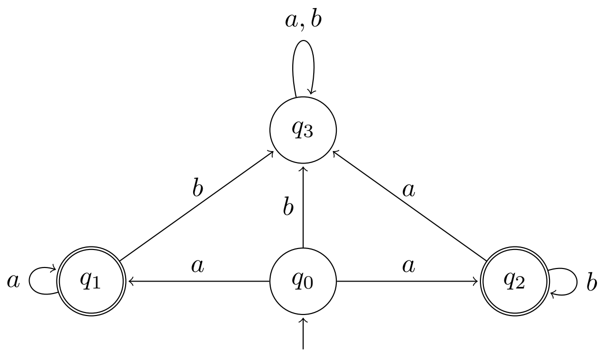


Figure 1 A nondeterministic Büchi word automaton over $\{a, b\}$. This NBW is complete and accepts the language $\{a^\omega, ab^\omega\}$.

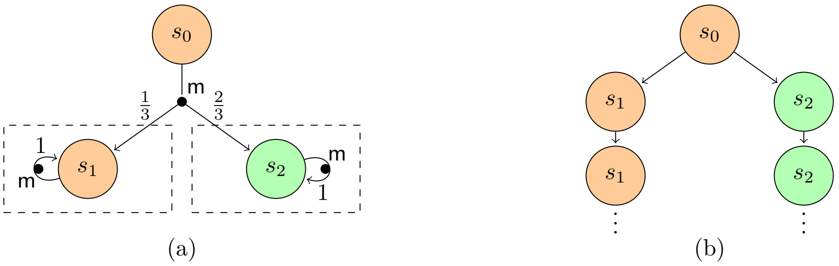


Figure 2 (a) An MDP with initial state s_0. The set of labels is $\{a, b\}$ and the labelling function for the MDP is as follows: $\ell(s_0) = \ell(s_1) = a$, $\ell(s_2) = b$. The labels are indicated by different colours. Since each state has only one available action m, the MDP is actually an MC. There are two end-components in this MDP labelled with the two dashed boxes. (b) The tree that stems from unravelling of the MC with initial state s_0 on the left, while disregarding probabilities.

of the complete word as a nondeterministic automaton normally would without changing their language. They can be characterised in many ways, including as automata that simulate deterministic automata. The NBW in Figure 1 is neither GFG nor good-for-MDPs (GFM) as shown later.

2.2 Markov Decision Processes (MDPs)

A *(finite, state-labelled) Markov decision process* (MDP) is a tuple $\langle S, \mathsf{Act}, \mathsf{P}, \Sigma, \ell\rangle$ consisting of a finite set S of states, a finite set Act of actions, a partial function $\mathsf{P} : S \times \mathsf{Act} \nrightarrow \mathrm{Distr}(S)$ denoting the probabilistic transition and a labelling function $\ell : S \to \Sigma$. The set of available actions in a state s is $\mathsf{Act}(s) = \{\mathsf{m} \in \mathsf{Act} \mid \mathsf{P}(s, \mathsf{m}) \text{ is defined}\}$. An MDP is a (labelled) Markov chain (MC) if $|\mathsf{Act}(s)| = 1$ for all $s \in S$.

An infinite *run (path)* of an MDP $\mathcal{M}$ is a sequence $s_0\mathsf{m}_1 \ldots \in (S \times \mathsf{Act})^\omega$ such that $\mathsf{P}(s_i, \mathsf{m}_{i+1})$ is defined and $\mathsf{P}(s_i, \mathsf{m}_{i+1})(s_{i+1}) > 0$ for all $i \geq 0$. A finite run is a finite such sequence. Let $\Omega(\mathcal{M})$ ($\mathrm{Paths}(\mathcal{M})$) denote the set of (finite) runs in $\mathcal{M}$ and $\Omega(\mathcal{M})_s$ ($\mathrm{Paths}(\mathcal{M})_s$) denote the set of (finite) runs in $\mathcal{M}$ starting from s. Abusing the notation slightly, for an infinite run $r = s_0\mathsf{m}_1s_1\mathsf{m}_2 \ldots$ we write $\ell(r) = \ell(s_0)\ell(s_1) \ldots \in \Sigma^\omega$.

A *strategy* for an MDP is a function $\mu : \mathrm{Paths}(\mathcal{M}) \to \mathrm{Distr}(\mathsf{Act})$ that, given a finite run r, returns a probability distribution on all the available actions at the last state of r. A *memoryless* (positional) strategy for an MDP is a function $\mu : S \to \mathrm{Distr}(\mathsf{Act})$ that, given a

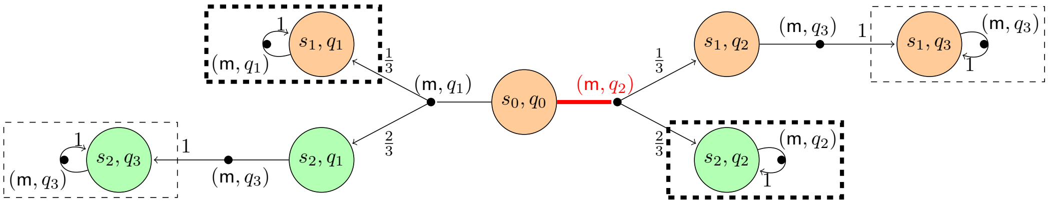


Figure 3 An example of a product MDP $\mathcal{M} \times \mathcal{N}$ with initial state (s_0, q_0) and $F^{\times} = \{(s_1, q_1), (s_2, q_1), (s_1, q_2), (s_2, q_2)\}$ where $\mathcal{M}$ is the MDP (MC) in Figure 2(a) and $\mathcal{N}$ is the NBW in Figure 1. The states $(s_0, q_0), (s_1, q_1), (s_1, q_2)$ and (s_1, q_3) are labelled with a while all the other states are labelled with b. Again, the four end-components of the MDP are labelled with dashed boxes; the upper left and lower right end-components are accepting (highlighted in thick dashed boxes).

state s, returns a probability distribution on all the available actions at that state. The set of runs $\Omega(\mathcal{M})^{\mu}_{s}$ is a subset of $\Omega(\mathcal{M})_s$ that correspond to strategy μ and initial state s. Given a memoryless/finite-memory strategy μ for $\mathcal{M}$, an MC $(\mathcal{M})_{\mu}$ is induced [3, Section 10.6].

A *sub-MDP* of $\mathcal{M}$ is an MDP $\mathcal{M}' = \langle S', \mathsf{Act}', \mathsf{P}', \Sigma, \ell' \rangle$, where $S' \subseteq S$, $\mathsf{Act}' \subseteq \mathsf{Act}$ is such that $\mathsf{Act}'(s) \subseteq \mathsf{Act}(s)$ for every $s \in S'$, and P' and ℓ' are analogous to P and ℓ when restricted to S' and Act'. In particular, $\mathcal{M}'$ is closed under probabilistic transitions, i.e. for all $s \in S'$ and $\mathsf{m} \in \mathsf{Act}'$ we have that $\mathsf{P}'(s, \mathsf{m})(s') > 0$ implies that $s' \in S'$. An *end-component* [1, 3] of an MDP $\mathcal{M}$ is a sub-MDP $\mathcal{M}'$ of $\mathcal{M}$ such that its underlying graph is strongly connected and it has no outgoing transitions. An example MDP is presented in Figure 2(a).

A strategy μ and an initial state $s \in S$ induce a standard probability measure on sets of infinite runs, see, e.g., [3]. Such measurable sets of infinite runs are called events or objectives. We write $\Pr^{\mu}_{s}$ for the probability of an event $E \subseteq sS^{\omega}$ of runs starting from s.

▶ **Theorem 1** (End-Component Properties [1, 3])**.** *Once an end-component E of an MDP is entered, there is a strategy that visits every state-action combination in E with probability 1 and stays in E forever. Moreover, for every strategy the union of the end-components is visited with probability 1.*

2.3 The Product of MDPs and Automata

Given an MDP $\mathcal{M} = \langle S, \mathsf{Act}, \mathsf{P}, \Sigma, \ell \rangle$ with initial state $s_0 \in S$ and an NBW $\mathcal{N} = \langle \Sigma, Q, \delta, q_0, F \rangle$, we want to compute an optimal strategy satisfying the objective that the run of $\mathcal{M}$ is in the language of $\mathcal{N}$. We define the semantic satisfaction probability for $\mathcal{N}$ and a strategy μ from state $s \in S$ as: $\mathrm{PSem}^{\mathcal{M}}_{\mathcal{N}}(s, \mu) = \Pr^{\mu}_{s}\{r \in \Omega^{\mu}_{s} : \ell(r) \in \mathcal{L}(\mathcal{N})\}$ and $\mathrm{PSem}^{\mathcal{M}}_{\mathcal{N}}(s) = \sup_{\mu} \big(\mathrm{PSem}^{\mathcal{M}}_{\mathcal{N}}(s, \mu)\big)$. In the case that $\mathcal{M}$ is an MC, we simply have $\mathrm{PSem}^{\mathcal{M}}_{\mathcal{N}}(s) = \Pr_s\{r \in \Omega_s : \ell(r) \in \mathcal{L}(\mathcal{N})\}$.

The *product* of $\mathcal{M}$ and $\mathcal{N}$ is an MDP $\mathcal{M} \times \mathcal{N} = \langle S \times Q, \mathsf{Act} \times Q, \mathsf{P}^{\times}, \Sigma, \ell^{\times} \rangle$ augmented with the initial state (s_0, q_0) and the Büchi acceptance condition $F^{\times} = \{(s, q) \in S \times Q \mid q \in F\}$. The labelling function $\ell^{\times}$ maps each state $(s, q) \in S \times Q$ to $\ell(s)$.

We define the partial function $\mathsf{P}^{\times} : (S \times Q) \times (\mathsf{Act} \times Q) \nrightarrow \mathrm{Distr}(S \times Q)$ as follows: for all $(s, \mathsf{m}) \in \mathsf{support}(\mathsf{P})$, $s' \in S$ and $q, q' \in Q$, we have $\mathsf{P}^{\times}\big((s, q), (\mathsf{m}, q')\big)\big((s', q')\big) = \mathsf{P}(s, \mathsf{m})(s')$ for all $q' \in \delta(q, \ell(s))$[1].

[1] When $\mathcal{N}$ is complete, there always exists a state q' such that $q' \in \delta(q, \ell(s))$.

We define the syntactic satisfaction probability for the product MDP and a strategy $\mu^\times$ from a state (s,q) as: $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}\big((s,q),\mu^\times\big) = \mathrm{Pr}_{s,q}^{\mu^\times}\{r \in \Omega_{s,q}^{\mu^\times} : \ell^\times(r) \in \mathcal{L}(\mathcal{N})\}$[2] and $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}(s) = \sup_{\mu^\times}\big(\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}\big((s,q_0),\mu^\times\big)\big)$. The set of actions is Act in the MDP $\mathcal{M}$ while it is $\mathsf{Act} \times Q$ in the product MDP. This makes PSem and PSyn potentially different. In general, $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}(s) \leq \mathrm{PSem}_{\mathcal{N}}^{\mathcal{M}}(s)$ for all $s \in S$, because accepting runs can only occur on accepted words. If $\mathcal{N}$ is deterministic, $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}(s) = \mathrm{PSem}_{\mathcal{N}}^{\mathcal{M}}(s)$ holds for all $s \in S$.

End-components and runs are defined for products just like for MDPs. A run of $\mathcal{M} \times \mathcal{N}$ is accepting if it satisfies the product's acceptance condition. An accepting end-component of $\mathcal{M} \times \mathcal{N}$ is an end-component which contains some states in $F^\times$.

An example of a product MDP is presented in Figure 3. It is the product of the MDP in Figure 2(a) and the NBW in Figure 1. Since $\ell(r)$ is in the language of the NBW for every run r of the MDP, we have $\mathrm{PSem}_{\mathcal{N}}^{\mathcal{M}}(s_0) = 1$. However, the syntactic satisfaction probability $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}(s_0) = \frac{2}{3}$ is witnessed by the memoryless strategy which chooses the action (m, q_2) at the initial state. We do not need to specify the strategy for the other states since there is only one available action for any remaining state. According to the following definition, the NBW in Figure 1 is not GFM as witnessed by the MDP in Figure 2(a).

▶ **Definition 2** ([9]). *An NBW $\mathcal{N}$ is good-for-MDPs (GFM) if, for all finite MDPs $\mathcal{M}$ with initial state s_0, $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}(s_0) = \mathrm{PSem}_{\mathcal{N}}^{\mathcal{M}}(s_0)$ holds.*

3 PSPACE-Hardness

We show that the problem of checking whether or not a given NBW is GFM is PSPACE-hard. The reduction is from the NFA universality problem, which is known to be PSPACE-complete [18]. Given an NFA $\mathcal{A}$, the NFA universality problem asks whether $\mathcal{A}$ accepts every string, that is, whether $\mathcal{L}(\mathcal{A}) = \Sigma^*$.

We first give an overview of how this reduction works. Given a complete NFA $\mathcal{A}$, we first construct an NBW $\mathcal{A}_f$ (Definition 4) which can be shown to be GFM (Lemma 6). Using this NBW $\mathcal{A}_f$, we then construct another NBW $\mathsf{fork}(\mathcal{A}_f)$ (Definition 7). We complete the argument by showing in Lemma 8 that the NBW $\mathsf{fork}(\mathcal{A}_f)$ is GFM if, and only if, $\mathcal{A}$ accepts the universal language.

We start with the small observation that "for all finite MDPs" in Definition 2 can be replaced by "for all finite MCs".

▶ **Theorem 3.** *An NBW $\mathcal{N}$ is GFM iff, for all finite MCs $\mathcal{M}$ with initial state s_0, $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}(s_0) = \mathrm{PSem}_{\mathcal{N}}^{\mathcal{M}}(s_0)$ holds.*

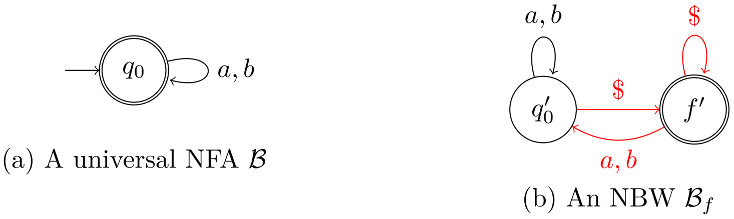
Proof.

"if": This is the case because there is an optimal finite memory control for an MDP $\mathcal{M}$, e.g. by using a language equivalent DPW $\mathcal{P}$ [13] and using its memory structure as finite memory. That is, we obtain an MC $\mathcal{M}'$ by applying an optimal memoryless strategy for $\mathcal{M} \times \mathcal{P}$ [4]. Naturally, if $\mathcal{N}$ satisfies the condition for $\mathcal{M}'$, then it also satisfies it for $\mathcal{M}$.

"only if": MCs are just special cases of MDPs. ◀

Given a complete NFA $\mathcal{A}$, we construct an NBW $\mathcal{A}_f$ by introducing a new letter \$ and a new state. As an example, given an NFA (DFA) $\mathcal{B}$ in Figure 4(a), we obtain an NBW $\mathcal{B}_f$ in Figure 4(b). It is easy to see that $\mathcal{L}(\mathcal{B}) = \Sigma^*$ where $\Sigma = \{a, b\}$.

[2] Let $\inf(r)$ be the set of states that appears infinite often in a run r. We also have $\mathrm{PSyn}_{\mathcal{N}}^{\mathcal{M}}((s,q),\mu^\times) = \mathrm{Pr}_{s,q}^{\mu^\times}\{r \in \Omega_{s,q}^{\mu^\times} : \inf(r) \cap F^\times \neq \emptyset\}$.



(a) A universal NFA $\mathcal{B}$

(b) An NBW $\mathcal{B}_f$

Figure 4 (a) $\mathcal{B}$ is a complete universal NFA. Let $\Sigma = \{a, b\}$. We have $\mathcal{L}(\mathcal{A}) = \Sigma^*$. (b) On the right is the corresponding complete NBW $\mathcal{B}_f$. The new final state f and the added transitions are highlighted in red. We have $\mathcal{L}(\mathcal{B}_f) = \{w_1\$w_2\$w_3\$\ldots\}$ where $w_i \in \Sigma^*$.

▶ **Definition 4.** *Given a complete NFA $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, we define the NBW $\mathcal{A}_f = (\Sigma_\$, Q_f, q_0, \delta_f, \{f\})$ with $\Sigma_\$ = \Sigma \cup \{\$\}$ and $Q_f = Q \cup \{f\}$ for a fresh letter $\$ \notin \Sigma$ and a fresh state $f \notin Q$, and with $\delta_f(q, \sigma) = \delta(q, \sigma)$ for all $q \in Q$ and $\sigma \in \Sigma$, $\delta_f(q, \$) = \{f\}$ for all $q \in F$, $\delta_f(q, \$) = \{q_0\}$ for all $q \in Q \setminus F$, and $\delta_f(f, \sigma) = \delta_f(q_0, \sigma)$ for all $\sigma \in \Sigma_\$$.*

The language of $\mathcal{A}_f$ consists of all words of the form $w_1\$w_1'\$w_2\$w_2'\$w_3\$w_3'\$\ldots$ such that, for all $i \in \mathbb{N}$, $w_i \in {\Sigma_\$}^*$ and $w_i' \in \mathcal{L}(\mathcal{A})$. This provides the following lemma.

▶ **Lemma 5.** *Given two NFAs $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$ if, and only if, $\mathcal{L}(\mathcal{B}_f) \subseteq \mathcal{L}(\mathcal{A}_f)$.*

The following lemma simply states that the automaton $\mathcal{A}_f$ from the above construction is GFM. This lemma is technical and is key to prove Lemma 8, the main lemma, of this section.

▶ **Lemma 6.** *For every complete NFA $\mathcal{A}$, $\mathcal{A}_f$ is GFM.*

Proof. Consider an arbitrary MC $\mathcal{M}$ with initial state s_0. We show that $\mathcal{A}_f$ is good for $\mathcal{M}$, that is, $\mathrm{PSem}^{\mathcal{M}}_{\mathcal{A}_f}(s_0) = \mathrm{PSyn}^{\mathcal{M}}_{\mathcal{A}_f}(s_0)$. It suffices to show $\mathrm{PSyn}^{\mathcal{M}}_{\mathcal{A}_f}(s_0) \geq \mathrm{PSem}^{\mathcal{M}}_{\mathcal{A}_f}(s_0)$ since by definition the converse $\mathrm{PSem}^{\mathcal{M}}_{\mathcal{A}_f}(s_0) \geq \mathrm{PSyn}^{\mathcal{M}}_{\mathcal{A}_f}(s_0)$ always holds.

First, we construct a language equivalent deterministic Büchi automaton (DBW) $\mathcal{D}_f$ by first determinising the NFA $\mathcal{A}$ to a DFA $\mathcal{D}$ by a standard subset construction and then obtain $\mathcal{D}_f$ by Definition 4. Since $\mathcal{L}(\mathcal{A}_f) = \mathcal{L}(\mathcal{D}_f)$, we have that $\mathrm{PSem}^{\mathcal{M}}_{\mathcal{A}_f}(s_0) = \mathrm{PSem}^{\mathcal{M}}_{\mathcal{D}_f}(s_0)$. In addition, since $\mathcal{D}_f$ is deterministic, we have $\mathrm{PSem}^{\mathcal{M}}_{\mathcal{D}_f}(s_0) = \mathrm{PSyn}^{\mathcal{M}}_{\mathcal{D}_f}(s_0)$.

It remains to show $\mathrm{PSyn}^{\mathcal{M}}_{\mathcal{A}_f}(s_0) \geq \mathrm{PSyn}^{\mathcal{M}}_{\mathcal{D}_f}(s_0)$. For that, it suffices to show that for an arbitrary accepting run r of $\mathcal{M} \times \mathcal{D}_f$, there is a strategy for $\mathcal{M} \times \mathcal{A}_f$ such that r' (the corresponding run in the product) is accepting in $\mathcal{M} \times \mathcal{A}_f$ where the projections of r and r' on $\mathcal{M}$ are the same.

Consider an accepting run of $\mathcal{M} \times \mathcal{D}_f$. Before entering an accepting end-component of $\mathcal{M} \times \mathcal{D}_f$, any strategy to resolve the nondeterminism in $\mathcal{M} \times \mathcal{A}_f$ (thus $\mathcal{A}_f$) can be used. This will not block $\mathcal{A}_f$, as it is a complete automaton, and $\mathcal{A}_f$ is essentially re-set whenever it reads a \$. Once an accepting end-component of $\mathcal{M} \times \mathcal{D}_f$ is entered, there must exist a word of the form $\$w\$$, where $w \in \mathcal{L}(\mathcal{D})$ (and thus $w \in \mathcal{L}(\mathcal{A})$), which is a possible initial sequence of letters produced from some state m of $\mathcal{M} \times \mathcal{D}_f$ in this end-component. We fix such a word $\$w\$$; such a state m of the end-component in $\mathcal{M} \times \mathcal{D}_f$ from which this word $\$w\$$ can be produced; and strategy of the NBW $\mathcal{A}_f$ to follow the word $w\$$ from q_0 (and f) to the accepting state f. (Note that the first \$ always leads to q_0 or f.)

In an accepting end-component we can be in two modes: *tracking*, or *not tracking*. If we are *not tracking* and reach m, we start to *track* $\$w\$$: we use $\mathcal{A}_f$ to reach an accepting state after reading $\$w\$$ (ignoring what happens in any other case) with the strategy we have

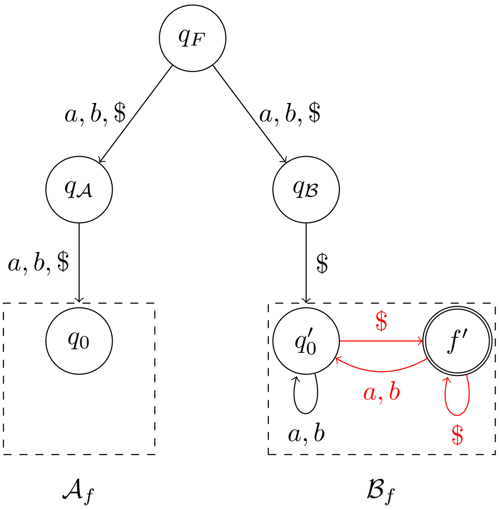


Figure 5 Given a complete NFA $\mathcal{A}$, an NBW $\mathcal{A}_f$ and an NBW $\mathsf{fork}(\mathcal{A}_f)$ are constructed. In this example, $\Sigma = \{a, b\}$ and $\Sigma_\$ = \{a, b, \$\}$. From the state $q_{\mathcal{A}}$ (resp. $q_{\mathcal{B}}$), the NBW $\mathsf{fork}(\mathcal{A}_f)$ transitions to the initial state of $\mathcal{A}_f$ (resp. $\mathcal{B}_f$).

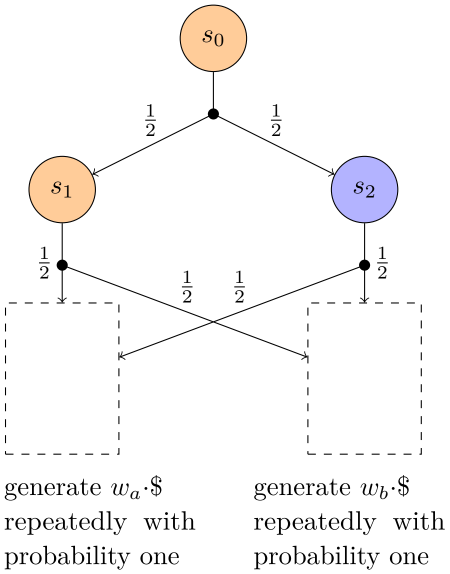

s_0
$\frac{1}{2}$
$\frac{1}{2}$
s_1
s_2
$\frac{1}{2}$
$\frac{1}{2}$
$\frac{1}{2}$
$\frac{1}{2}$
generate $w_a{\cdot}\$$ repeatedly with probability one
generate $w_b{\cdot}\$$ repeatedly with probability one

Figure 6 An example MC in the proof of Lemma 8. Assume $\Sigma_\$ = \{a, b, \$\}$. The labelling of the MC is as follows: $\ell(s_0) = \ell(s_1) = a$ and $\ell(s_2) = \$$.

fixed. Note that, after reading the first \$, we are in either q_0 or f, such that, when starting from m, it is always possible, with a fixed probability $p > 0$, to read $\$w\$$, and thus to accept. If we read an unexpected letter (where the "expected" letter is always the next letter from $\$w\$$) or the end of the word $\$w\$$ is reached, we move to *not tracking*.

The automata choices when *not tracking* can be resolved arbitrarily.

Once in an accepting end-component of $\mathcal{M} \times \mathcal{D}_f$, tracking is almost surely started infinitely often, and it is thus almost surely successful infinitely often. Thus, we have $\mathrm{PSyn}^{\mathcal{M}}_{\mathcal{A}_f}(s_0) \geq \mathrm{PSyn}^{\mathcal{M}}_{\mathcal{D}_f}(s_0)$. ◀

Let $\mathcal{B}$ be an universal NFA in Figure 4(a) and $\mathcal{B}_f = (\Sigma_\$, Q'_f, q'_0, \delta'_f, \{f'\})$ be the NBW in Figure 4(b). Assume without loss of generality that the state space of $\mathcal{A}_f$, $\mathcal{B}_f$, and $\{q_F, q_{\mathcal{A}}, q_{\mathcal{B}}\}$ are pairwise disjoint. We now define the fork operation. An example of how to construct an NBW $\mathsf{fork}(\mathcal{A}_f)$ is shown in Figure 5.

▶ **Definition 7.** *Given an NBW $\mathcal{A}_f = (\Sigma_\$, Q_f, q_0, \delta_f, \{f\})$, we define the NBW $\mathsf{fork}(\mathcal{A}_f) = (\Sigma_\$, Q_F, q_F, \delta_F, \{f, f'\})$ with*

- *$Q_F = Q_f \cup Q'_{f'} \cup \{q_F, q_{\mathcal{A}}, q_{\mathcal{B}}\}$;*
- *$\delta_F(q, \sigma) = \delta_f(q, \sigma)$ for all $q \in Q_f$ and $\sigma \in \Sigma_\$$;*
- *$\delta_F(q, \sigma) = \delta_{f'}(q, \sigma)$ for all $q \in Q'_{f'}$ and $\sigma \in \Sigma_\$$;*
- *$\delta_F(q_F, \sigma) = \{q_{\mathcal{A}}, q_{\mathcal{B}}\}$ for all $\sigma \in \Sigma_\$$;*
- *$\delta_F(q_{\mathcal{A}}, \sigma) = \{q_0\}$ for all $\sigma \in \Sigma_\$$;*
- *$\delta_F(q_{\mathcal{B}}, \$) = \{q'_0\}$, and $\delta_F(q_{\mathcal{B}}, \sigma) = \emptyset$ for all $\sigma \in \Sigma$.*

Following from Lemma 5 and Lemma 6, we have:

▶ **Lemma 8.** *The NBW $\mathsf{fork}(\mathcal{A}_f)$ is GFM if, and only if, $\mathcal{L}(\mathcal{A}) = \Sigma^*$.*

Proof. We first observe that $\mathcal{L}\big(\mathsf{fork}(\mathcal{A}_f)\big) = \{\sigma\sigma' w \mid \sigma, \sigma' \in \Sigma_\$,\ w \in \mathcal{L}(\mathcal{A}_f)\} \cup \{\sigma\$w \mid \sigma \in \Sigma_\$,\ w \in \mathcal{L}(\mathcal{B}_f)\}$.

"if": When $\mathcal{L}(\mathcal{A}) = \Sigma^* = \mathcal{L}(\mathcal{B})$ holds, Lemma 5 provides $\{\sigma\sigma' w \mid \sigma, \sigma' \in \Sigma_\$,\ w \in \mathcal{L}(\mathcal{A}_f)\} \supset \{\sigma \$ w \mid \sigma \in \Sigma_\$,\ w \in \mathcal{L}(\mathcal{B}_f)\}$, and therefore $\mathcal{L}(\mathsf{fork}(\mathcal{A}_f)) = \{\sigma\sigma' w \mid \sigma, \sigma' \in \Sigma_\$,\ w \in \mathcal{L}(\mathcal{A}_f)\}$.
As $\mathcal{A}_f$ is GFM by Lemma 6, this provides the GFM strategy "move first to $q_\mathcal{A}$, then to q_0, and henceforth follow the GFM strategy of $\mathcal{A}_f$ for $\mathsf{fork}(\mathcal{A}_f)$". Thus, $\mathsf{fork}(\mathcal{A}_f)$ is GFM in this case.

"only if": Assume $\mathcal{L}(\mathcal{A}) \neq \Sigma^* = \mathcal{L}(\mathcal{B})$, that is, $\mathcal{L}(\mathcal{A}) \subset \mathcal{L}(\mathcal{B})$. There must exist words $w_a \in \mathcal{L}(\mathcal{A})$ and $w_b \in \mathcal{L}(\mathcal{B}) \setminus \mathcal{L}(\mathcal{A})$. We now construct an MC which witnesses that $\mathsf{fork}(\mathcal{A}_f)$ is not GFM.

The MC emits an a at the first step and then either an a or a \$ with a chance of $\frac{1}{2}$ at the second step. An example is provided in Figure 6.

After these two letters, it then moves to one of two cycles (independent of the first two chosen letters) with equal chance of $\frac{1}{2}$; one of these cycles repeats a word $w_a{\cdot}\$$ infinitely often, while the other repeats a word $w_b{\cdot}\$$ infinitely often, where $w_a \in \mathcal{L}(\mathcal{A})$.

It is easy to see that the semantic chance of acceptance is $\frac{3}{4}$ – failing if, and only if, the second letter is a and the word $w_b\$$ is subsequently repeated infinitely often – whereas the syntactic chance of satisfaction is $\frac{1}{2}$: when the automaton first moves to $q_\mathcal{A}$, it accepts if, and only if, the word $w_a\$$ is later repeated infinitely often, which happens with a chance of $\frac{1}{2}$; when the automaton first moves to $q_\mathcal{B}$, it will reject when \$ is not the second letter, which happens with a chance of $\frac{1}{2}$. ◀

It follows from Lemma 8 that the NFA universality problem is polynomial-time reducible to the problem of whether or not a given NBW is GFM.

▶ **Theorem 9.** *The problem of whether or not a given NBW is GFM is* PSPACE*-hard.*

Using the same construction of Definition 4, we can show that the problem of minimising a GFM NBW is PSPACE-hard. The reduction is from a problem which is similar to the NFA universality problem.

▶ **Theorem 10.** *Given a GFM NBW and a bound k, the problem whether there is a language equivalent GFM NBW with at most k states is* PSPACE*-hard. It is* PSPACE*-hard even for (fixed) $k = 2$.*

Proof. Using the construction of Definition 4, PSPACE-hardness follows from a reduction from the problem whether a nonempty complete NFA accepts all nonempty words. The latter problem is PSPACE-complete, following the PSPACE-completeness of the universality problem of (general) NFAs [18].

The reduction works because, for a nonempty complete NFA A the following hold:

(a) A GFM NBW equivalent to $\mathcal{A}_f$ must have at least 2 states, one final and one nonfinal. This is because it needs a final state (as some word is accepted) as well as a nonfinal one (words that contain finitely many \$s are rejected).

(b) For a 2-state minimal GFM NBW equivalent to $\mathcal{A}_f$, there cannot be a word $w \in \Sigma^+$ that goes from the final state back to it as this would produce an accepting run with finitely many \$s (as there is some accepting run). Therefore, when starting from the final state, any finite word can only go to the nonfinal state and stay there or block. But blocking is no option, as there is an accepted continuation to an infinite word. Thus, all letters in Σ lead from either state to the nonfinal state (only).

In order for a word starting with a letter in Σ to be accepted, there must therefore be a \$ transition from the nonfinal to the final state.

These two points imply that for a nonempty complete NFA $\mathcal{A}$ such that there is a 2-state GFM NBW equivalent to $\mathcal{A}_f$ iff $\mathcal{A}$ accepts all nonempty words. ◀

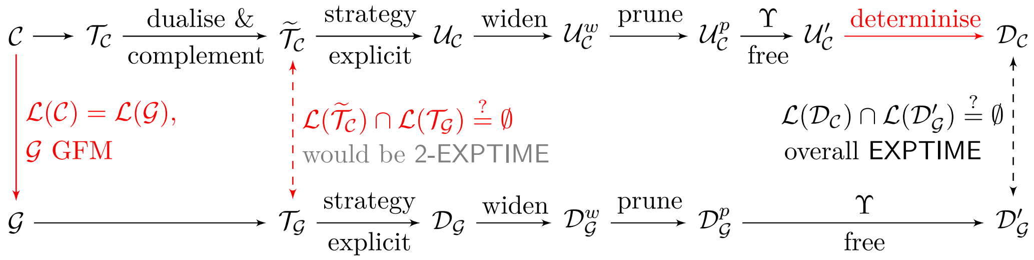

$\mathcal{C}$
$\mathcal{T}_\mathcal{C}$
dualise &
complement
$\widetilde{\mathcal{T}}_\mathcal{C}$
strategy
explicit
$\mathcal{U}_\mathcal{C}$
widen
$\mathcal{U}^w_\mathcal{C}$
prune
$\mathcal{U}^p_\mathcal{C}$
Υ
free
$\mathcal{U}'_\mathcal{C}$
determinise
$\mathcal{D}_\mathcal{C}$
$\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{G})$,
$\mathcal{G}$ GFM
$\mathcal{L}(\widetilde{\mathcal{T}}_\mathcal{C}) \cap \mathcal{L}(\mathcal{T}_\mathcal{G}) \overset{?}{=} \emptyset$
would be 2-EXPTIME
$\mathcal{L}(\mathcal{D}_\mathcal{C}) \cap \mathcal{L}(\mathcal{D}'_\mathcal{G}) \overset{?}{=} \emptyset$
overall EXPTIME
$\mathcal{G}$
$\mathcal{T}_\mathcal{G}$
strategy
explicit
$\mathcal{D}_\mathcal{G}$
widen
$\mathcal{D}^w_\mathcal{G}$
prune
$\mathcal{D}^p_\mathcal{G}$
Υ
free
$\mathcal{D}'_\mathcal{G}$

Figure 7 Flowchart of the algorithms. The first algorithm is in 2-EXPTIME, which is to check the nonemptiness of the intersection of $\widetilde{\mathcal{T}}_\mathcal{C}$ and $\mathcal{T}_\mathcal{G}$. The second algorithm is in EXPTIME, which is to check the nonemptiness of the intersection of $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}'_\mathcal{G}$. The steps that have exponential blow-up are highlighted in red.

4 Decision Procedure for Qualitative GFM

In this section, we first define the notion of qualitative GFM (QGFM) and then provide an EXPTIME procedure that decides QGFM-ness.

The definition of QGFM is similar to GFM but we only need to consider MCs with which the semantic chance of success is one:

▶ **Definition 11.** *An NBW $\mathcal{N}$ is qualitative good-for-MDPs (QGFM) if, for all finite MDPs $\mathcal{M}$ with initial state s_0 and $\mathrm{PSem}^\mathcal{M}_\mathcal{N}(s_0) = 1$, $\mathrm{PSyn}^\mathcal{M}_\mathcal{N}(s_0) = 1$ holds.*

Similar to Theorem 3, we can also replace "MDPs" by "MCs" in the definiton of QGFM:

▶ **Theorem 12.** *An NBW $\mathcal{N}$ is QGFM iff, for all finite MCs $\mathcal{M}$ with initial state s_0 and $\mathrm{PSem}^\mathcal{M}_\mathcal{N}(s_0) = 1$, $\mathrm{PSyn}^\mathcal{M}_\mathcal{N}(s_0) = 1$ holds.*

To decide QGFM-ness, we make use of the well known fact that qualitative acceptance, such as $\mathrm{PSem}^\mathcal{M}_\mathcal{N}(s_0) = 1$, does not depend on the probabilities for an MC $\mathcal{M}$. This can, for example, be seen by considering the syntactic product $\mathrm{PSyn}^\mathcal{M}_\mathcal{D}(s_0) = 1$ with a deterministic parity automaton $\mathcal{D}$ (for a deterministic automaton, $\mathrm{PSem}^\mathcal{M}_\mathcal{D}(s_0) = \mathrm{PSyn}^\mathcal{M}_\mathcal{D}(s_0)$ trivially holds), where changing the probabilities does not change the end-components of the product MC $\mathcal{M} \times \mathcal{D}$, and the acceptance of these end-components is solely determined by the highest colour of the states (or transitions) occurring in it, and thus also independent of the probabilities: the probability is 1 if, and only if, an accepting end-component can be reached almost surely, which is also independent of the probabilities. As a result, we can search for the (regular) tree that stems from the unravelling of an MC, while disregarding probabilities. See Figure 2(b) for an example of such a tree.

This observation has been used in the synthesis of probabilistic systems before [15]. The set of directions (of a tree) Υ could then, for example, be chosen to be the set of states of the unravelled finite MC; this would not normally be a full tree.

In the following, we show an exponential-time algorithm to decide whether a given NBW is QGFM or not. This procedure involves transformations of tree automata with different acceptance conditions. Because this is quite technical, we only provide an outline in the main paper. More notations (Section A.1) and details of the constructions (Section A.2) are provided in the full version [16].

For a given candidate NBW $\mathcal{C}$, we first construct a language equivalent NBW $\mathcal{G}$ that we know to be GFM, such as the slim automaton from [9] or a suitable limit deterministic automaton [21, 6, 7, 17, 8]. For all known constructions, $\mathcal{G}$ can be exponentially larger than $\mathcal{C}$. We use the slim automata from [9]; they have $O(3^{|Q|})$ states and transitions.

We then construct a number of tree automata as outlined in Figure 7. In a first construction, we discuss in the full version [16] how to build, for an NBW $\mathcal{N}$, a (symmetric) alternating Büchi tree automaton (ABS) $\mathcal{T}_\mathcal{N}$ that accepts (the unravelling of) an MC (without probabilities, as discussed above) $\mathcal{M}$ if, and only if, the syntactic product of $\mathcal{N}$ and $\mathcal{M}$ almost surely accepts. This construction is used twice: once to produce $\mathcal{T}_\mathcal{G}$ for the GFM automaton $\mathcal{G}$ we have constructed, and once to produce $\mathcal{T}_\mathcal{C}$ for our candidate automaton $\mathcal{C}$. Since $\mathcal{G}$ is QGFM, $\mathcal{T}_\mathcal{G}$ accepts all the MCs $\mathcal{M}$ that almost surely produce a run in $\mathcal{L}(\mathcal{G})$ (which is the same as $\mathcal{L}(\mathcal{C})$), that is, $\mathrm{PSem}_\mathcal{G}^\mathcal{M}(s_0) = \mathrm{PSem}_\mathcal{C}^\mathcal{M}(s_0) = 1$.

Therefore, to check whether or not our candidate NBW $\mathcal{C}$ is QGFM, we can test language equivalence of $\mathcal{T}_\mathcal{C}$ and $\mathcal{T}_\mathcal{G}$, e.g. by first complementing $\mathcal{T}_\mathcal{C}$ to $\widetilde{\mathcal{T}_\mathcal{C}}$ and then checking whether or not $\mathcal{L}(\widetilde{\mathcal{T}_\mathcal{C}}) \cap \mathcal{L}(\mathcal{T}_\mathcal{G}) = \emptyset$ holds: the MCs in the intersection of the languages witness that $\mathcal{C}$ is not QGFM. Thus, $\mathcal{C}$ is QGFM if, and only if, these languages do not intersect, that is, $\mathcal{L}(\widetilde{\mathcal{T}_\mathcal{C}}) \cap \mathcal{L}(\mathcal{T}_\mathcal{G}) = \emptyset$. This construction leads to a 2-EXPTIME procedure for deciding QGFM: we get the size of the larger automaton ($\mathcal{G}$) and the complexity of the smaller automaton $\mathcal{C}$. The purpose of the following delicate construction is to contain the exponential cost to the syntactic material of the smaller automaton, while still obtaining the required level of entanglement between the structures and retaining the size advantage from the GFM property of $\mathcal{G}$.

Starting from $\widetilde{\mathcal{T}_\mathcal{C}}$, we make a few transformations by mainly controlling the number of directions the alternating tree automaton needs to consider and the set of decisions player *accept* [3] has to make. This restricts the scope in such a way that the resulting intersection might shrink, but cannot become empty[4].

We rein in the number of directions in two steps: in a first step, we *increase* the number of directions by widening the run tree with one more direction than the size of the state space of the candidate automaton $\mathcal{C}$. The larger amount of directions allows us to concurrently untangle the decisions of player *accept* within and between $\widetilde{\mathcal{T}_\mathcal{C}}$ and $\mathcal{T}_\mathcal{G}$, which intuitively creates one distinguished direction for each state q of $\widetilde{\mathcal{T}_\mathcal{C}}$ used by player *accept*, and one (different) distinguished direction for $\mathcal{T}_\mathcal{G}$. In a second step, we only keep these directions, resulting in an automaton with a *fixed* branching degree (just one bigger than the size of the state space of $\mathcal{C}$), which is easy to analyse with standard techniques.

The standard techniques mean to first make the remaining choices of player *accept* in $\widetilde{\mathcal{T}_\mathcal{C}}$ explicit, which turns it into a universal co-Büchi automaton ($\mathcal{U}_\mathcal{C}$). The automaton is then simplified to the universal co-Büchi automaton $\mathcal{U}'_\mathcal{C}$ which can easily be determinised to a deterministic parity automaton $\mathcal{D}_\mathcal{C}$.

For $\mathcal{T}_\mathcal{G}$, a sequence of similar transformations are made; however, as we do not need to complement here, the automaton obtained from making the decisions explicit is already deterministic, which saves the exponential blow-up obtained in the determinisation of a universal automaton (determinising $\mathcal{U}'_\mathcal{C}$ to $\mathcal{D}_\mathcal{C}$).

Therefore, $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}'_\mathcal{G}$ can both be constructed from $\mathcal{C}$ in time exponential in $\mathcal{C}$, and checking their intersection for emptiness can be done in time exponential in $\mathcal{C}$, too. With that, we obtain the membership in EXPTIME for QGFM-ness:

▶ **Theorem 13.** *The problem of whether or not a given NBW is QGFM is in* EXPTIME*.*

[3] The acceptance of a tree by a tree automaton can be viewed as the outcome of a game played by player *accept* and player *reject*. We refer to the full version [16] for details.

[4] It can be empty to start with, of course, and will stay empty in that case. But if the intersection is not empty, then these operations will leave something in the language.

5 Membership in EXPTIME for GFM

In this section, we start out with showing a sufficient (Lemma 14) and necessary (Lemma 15) criterion for a candidate NBW to be GFM in Section 5.1.

We show in Section 5.2 that this criterion is also sufficient and necessary for QGFM-ness. This implies that GFM-ness and QGFM-ness collapse, so that the EXPTIME decision procedure from Section 4 can be used to decide GFM-ness, and the PSPACE hardness from Section 3 extends to QGFM.

5.1 Key Criterion for GFM-ness

In order to establish a necessary and sufficient criterion for GFM-ness, we construct two safety automata[5] $\mathcal{S}$ and $\mathcal{T}$.

Given a candidate NBW $\mathcal{C}$, we define some notions for the states and transitions. We say a state q of $\mathcal{C}$ is productive if $\mathcal{L}(\mathcal{C}_q) \neq \emptyset$ where $\mathcal{C}_q$ is the automaton obtained from $\mathcal{C}$ by making q the initial state. A state q of the NBW $\mathcal{C}$ is called QGFM if the automaton $\mathcal{C}_q$ is QGFM. A transition (q, σ, r) is called residual if $\mathcal{L}(\mathcal{C}_r) = \sigma^{-1}\mathcal{L}(\mathcal{C}_q)$ [11, 2]. In general, $\mathcal{L}(\mathcal{C}_r) \subseteq \sigma^{-1}\mathcal{L}(\mathcal{C}_q)$ holds. See Figure 1 for an example of non-residual transitions. Selecting either of the two transitions from q_0 will lose language: when selecting the transition to the left, the word $a \cdot b^\omega$ is no longer accepted. Likewise, when selecting the transition to the right, the word a^ω is no longer accepted. Thus, this automaton cannot make the decision to choose the left or the right transition, and neither (q_0, a, q_1) nor (q_0, a, q_2) is a residual transition.

Now we are ready to define $\mathcal{S}$ and $\mathcal{T}$. In the NBW $\mathcal{S}$, we include the states from the candidate NBW $\mathcal{C}$ that are productive and QGFM at the same time. We only include the residual transitions (in $\mathcal{C}$) between those states. In the NBW $\mathcal{T}$, we include only the productive states of $\mathcal{C}$ and the transitions between them. We then make both $\mathcal{S}$ and $\mathcal{T}$ safety automata by marking all states final. We first show that the criterion, $\mathcal{L}(\mathcal{S}) = \mathcal{L}(\mathcal{T})$ and $\mathcal{S}$ is GFG[6], is sufficient for $\mathcal{C}$ to be GFM. Similar to the proof of Lemma 6, to show the NBW $\mathcal{C}$ is GFM, we show there exists a strategy for $\mathcal{C}$ such that the syntactic and semantic chance of winning are the same for any MC.

▶ **Lemma 14.** *If $\mathcal{L}(\mathcal{S}) = \mathcal{L}(\mathcal{T})$ and $\mathcal{S}$ is GFG, then the candidate NBW $\mathcal{C}$ is GFM.*

Proof. As $\mathcal{T}$ contains all states and transitions from $\mathcal{S}$, $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{T})$ always holds. We assume that $\mathcal{L}(\mathcal{S}) \supseteq \mathcal{L}(\mathcal{T})$ holds and $\mathcal{S}$ is GFG.

By Theorem 3, to show $\mathcal{C}$ is GFM, it suffices to show that $\mathcal{C}$ is good for an arbitrary MC $\mathcal{M}$ with initial state s_0. We first determinise $\mathcal{C}$ to a DPW $\mathcal{D}$ [13]. Since $\mathcal{D}$ is deterministic and $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{C})$, we have $\mathrm{PSyn}_{\mathcal{D}}^{\mathcal{M}}(s_0) = \mathrm{PSem}_{\mathcal{D}}^{\mathcal{M}}(s_0) = \mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}}(s_0)$. Since $\mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}}(s_0) \geq \mathrm{PSyn}_{\mathcal{C}}^{\mathcal{M}}(s_0)$ always holds, we establish the equivalence of syntactic and semantic chance of winning for $\mathcal{M} \times \mathcal{C}$ by proving $\mathrm{PSyn}_{\mathcal{C}}^{\mathcal{M}}(s_0) \geq \mathrm{PSyn}_{\mathcal{D}}^{\mathcal{M}}(s_0) = \mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}}(s_0)$.

For that, we show for an arbitrary accepting run r of $\mathcal{M} \times \mathcal{D}$, there is a strategy for $\mathcal{M} \times \mathcal{C}$ such that r' (the corresponding run in the product) is accepting in $\mathcal{M} \times \mathcal{C}$ where the projections of r and r' on $\mathcal{M}$ are the same.

[5] A safety automaton is one where all states are final. These automata can be viewed as NFAs where convenient.

[6] GFG as a general property is tricky, but $\mathcal{S}$ is a safety automaton, and GFG safety automata are, for example, determinisable by pruning, and the property whether or not a safety automaton is GFG can be checked in polynomial time [5].

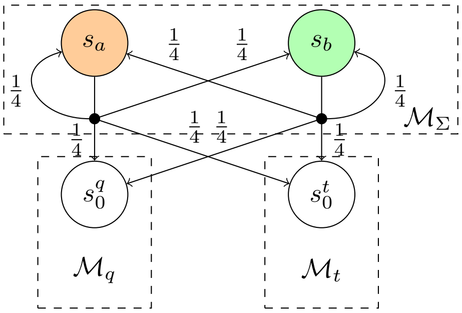


Figure 8 An example MC in the proof of Lemma 15. In this example, $\Sigma = \{a, b\}$. Also, the state q of the candidate NBW $\mathcal{C}$ is the only state that is not QGFM and the transition t of the NBW $\mathcal{C}$ is the only non-residual transition. We have $\ell(s_a) = a$ and $\ell(s_b) = b$. The states s_a and s_b and the transitions between them form $\mathcal{M}_\Sigma$.

We define the strategy for $\mathcal{M} \times \mathcal{C}$ depending on whether an accepting end-component of $\mathcal{M} \times \mathcal{D}$ has been entered. Since r is accepting, it must enter an accepting end-component of $\mathcal{M} \times \mathcal{D}$ eventually. Let the run r be $(s_0, q_0^{\mathcal{D}})(s_1, q_1^{\mathcal{D}}) \cdots$ and it enters the accepting end-component on reaching the state $(s_n, q_n^{\mathcal{D}})$. Before r enters an accepting end-component of $\mathcal{M} \times \mathcal{D}$, $\mathcal{C}$ follows the GFG strategy for $\mathcal{S}$ to stay within the states that are productive and QGFM. Upon reaching an accepting end-component of $\mathcal{M} \times \mathcal{D}$, the run r is in state $(s_n, q_n^{\mathcal{D}})$, assume the run for $\mathcal{M} \times \mathcal{C}$ is in state $(s_n, q_n^{\mathcal{C}})$ at this point. We then use the QGFM strategy of $\mathcal{C}_{q_n^{\mathcal{C}}}$ from here since $q_n^{\mathcal{C}}$ is QGFM.

We briefly explain why this strategy for $\mathcal{M} \times \mathcal{C}$ would lead to $\mathrm{PSyn}_{\mathcal{C}}^{\mathcal{M}}(s_0) \geq \mathrm{PSyn}_{\mathcal{D}}^{\mathcal{M}}(s_0)$. Since $(s_n, q_n^{\mathcal{D}})$ is in the accepting end-component of $\mathcal{M} \times \mathcal{D}$, we have $\mathrm{PSem}_{\mathcal{D}_{q_n^{\mathcal{D}}}}^{\mathcal{M}}(s_n) = \mathrm{PSyn}_{\mathcal{D}_{q_n^{\mathcal{D}}}}^{\mathcal{M}}(s_n) = 1$ by Theorem 1. We show $\mathrm{PSem}_{\mathcal{C}_{q_n^{\mathcal{C}}}}^{\mathcal{M}}(s_n) = 1$ so that $\mathrm{PSyn}_{\mathcal{C}_{q_n^{\mathcal{C}}}}^{\mathcal{M}}(s_n) = \mathrm{PSem}_{\mathcal{C}_{q_n^{\mathcal{C}}}}^{\mathcal{M}}(s_n) = 1$ as $q_n^{\mathcal{C}}$ is QGFM. For that, it suffices to show $\mathcal{L}(\mathcal{C}_{q_n^{\mathcal{C}}}) = \mathcal{L}(\mathcal{D}_{q_n^{\mathcal{D}}}) = w^{-1}\mathcal{L}(\mathcal{C})$ where $w = \ell(s_0 s_1 \cdots s_{n-1})$. This can be proved by induction in [16] over the length of the prefix of words from $\mathcal{L}(\mathcal{S})$. ◀

In order to show that this requirement is also necessary, we build an MC witnessing that $\mathcal{C}$ is not GFM in case the criterion is not satisfied. An example MC is given in Figure 8. We produce the MC by parts. It has a central part denoted by $\mathcal{M}_\Sigma$. The state space of $\mathcal{M}_\Sigma$ is S_Σ where $S_\Sigma = \{s_\sigma \mid \sigma \in \Sigma\}$. Each state s_σ is labelled with σ and there is a transition between every state pair.

For every state q that is not QGFM, we construct an MC $\mathcal{M}_q = \langle S_q, P_q, \Sigma, \ell_q \rangle$ from Section 4 witnessing that $\mathcal{C}_q$ is not QGFM, that is, from a designated initial state s_0^q, $\mathrm{PSyn}_{\mathcal{C}_q}^{\mathcal{M}_q}(s_0^q) \neq \mathrm{PSem}_{\mathcal{C}_q}^{\mathcal{M}_q}(s_0^q) = 1$, and for every non-residual transition $t = (q, \sigma, r)$ that is not in $\mathcal{S}$ due to $\mathcal{L}(\mathcal{C}_r) \neq \sigma^{-1}\mathcal{L}(\mathcal{C}_q)$, we construct an MC $\mathcal{M}_t = \langle S_t, P_t, \Sigma, \ell_t \rangle$ such that, from an initial state s_0^t, there is only one ultimately periodic word w_t produced, such that $w_t \in \sigma^{-1}\mathcal{L}(\mathcal{C}_q) \setminus \mathcal{L}(\mathcal{C}_r)$.

Finally, we produce an MC $\mathcal{M}$, whose states are the disjoint union of the MCs $\mathcal{M}_\Sigma$, $\mathcal{M}_q$ and $\mathcal{M}_t$ from above. The labelling and transitions within the MCs $\mathcal{M}_q$ and $\mathcal{M}_t$ are preserved while, from the states in S_Σ, $\mathcal{M}$ also transitions to all initial states of the individual $\mathcal{M}_q$ and $\mathcal{M}_t$ from above. It remains to specify the probabilities for the transitions from S_Σ: any state in S_Σ transitions to its successors uniformly at random.

▶ **Lemma 15.** *If $\mathcal{L}(\mathcal{S}) \neq \mathcal{L}(\mathcal{T})$ or $\mathcal{S}$ is not GFG, the candidate NBW $\mathcal{C}$ is not GFM.*

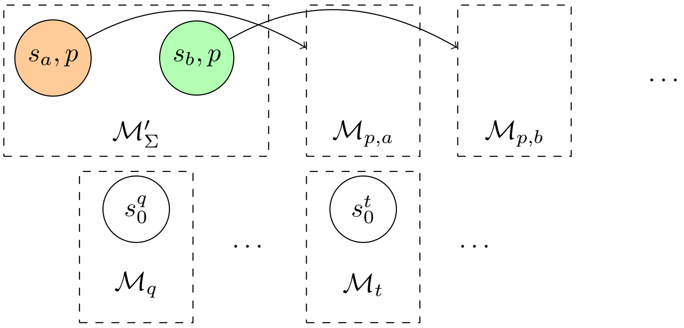


Figure 9 An illustration of the MC in the proof of Lemma 16. In this example, we have $\Sigma = \{a, b\}$. The new central part $\mathcal{M}'_\Sigma$ is obtained by removing the states that have no outgoing transitions in the cross product of $\mathcal{M}_\Sigma$ and $\mathcal{D}'$. For the states (s_a, p) and (s_b, p) of $\mathcal{M}'_\Sigma$, we have $\ell\big((s_a, p)\big) = a$ and $\ell\big((s_b, p)\big) = b$. For each state (s_σ, p) of the central part, we construct an MC $\mathcal{M}_{p,\sigma}$ using $\mathcal{D}'$. There is a transition from each state (s_σ, p) of $\mathcal{M}'_\Sigma$ to the initial state of $\mathcal{M}_{p,\sigma}$. The MCs $\mathcal{M}_q$ and $\mathcal{M}_t$ are as before. Whether there is a transition from a state from $\mathcal{M}'_\Sigma$ to the MCs $\mathcal{M}_q$ and $\mathcal{M}_t$ is determined by the overestimation provided by $\mathcal{D}'$.

Proof. Assume $\mathcal{L}(\mathcal{S}) \neq \mathcal{L}(\mathcal{T})$. There must exist a word $w = \sigma_0, \sigma_1, \ldots \in \mathcal{L}(\mathcal{T}) \setminus \mathcal{L}(S)$.

Let us use $\mathcal{M}$ with initial state s_{σ_0} as the MC which witnesses that $\mathcal{C}$ is not GFM. We first build the product MDP $\mathcal{M}\times\mathcal{C}$. There is a non-zero chance that, no matter how the choices of $\mathcal{C}$ (thus, the product MDP $\mathcal{M}\times\mathcal{C}$) are resolved, a state sequence $(s_{\sigma_0}, q_0), (s_{\sigma_1}, q_1), \ldots, (s_{\sigma_k}, q_k)$ with $k \geq 0$ is seen, and $\mathcal{C}$ selects a successor q such that (q_k, σ_k, q) is not a transition in $\mathcal{S}$.

For the case that this is because $\mathcal{C}_q$ is not QGFM, we observe that there is a non-zero chance that the product MDP moves to (s_0^q, q), such that $\mathrm{PSyn}_{\mathcal{C}}^{\mathcal{M}}(s_{\sigma_0}) < \mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}}(s_{\sigma_0})$ follows.

For the other case that this is because the transition $t = (q_k, \sigma_k, q)$ is non-residual, that is, $\mathcal{L}(\mathcal{C}_q) \neq {\sigma_k}^{-1}\mathcal{L}(\mathcal{C}_{q_k})$, we observe that there is a non-zero chance that the product MDP moves to (s_0^t, q), such that $\mathrm{PSyn}_{\mathcal{C}}^{\mathcal{M}}(s_{\sigma_0}) < \mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}}(s_{\sigma_0})$ follows.

For the case that $\mathcal{S}$ is not GFG, no matter how the nondeterminism of $\mathcal{C}$ is resolved, there must be a shortest word $w = \sigma_0, \ldots, \sigma_k$ $(k \geq 0)$ such that w is a prefix of a word in $\mathcal{L}(\mathcal{S})$, but the selected control leaves $\mathcal{S}$. For this word, we can argue in the same way as above. ◀

Lemma 14 and Lemma 15 suggest that GFM-ness of a NBW can be decided in EXPTIME by checking whether the criterion holds or not. However, as shown in the next section that QGFM = GFM, we can apply the EXPTIME procedure from Section 5 to check QGFM-ness, and thus, GFM-ness.

5.2 QGFM = GFM

To show that QGFM = GFM, we show that the same criterion from the previous section is also sufficient and necessary for QGFM. By definition, if an NBW is GFM then it is QGFM. Thus, the sufficiency of the criterion follows from Lemma 14. We are left to show the necessity of the criterion. To do that, we build an MC $\mathcal{M}'$ witnessing that $\mathcal{C}$ is not QGFM in case the criterion of Lemma 15 is not satisfied. We sketch in Figure 9 the construction of $\mathcal{M}'$.

The principle difference between the MC $\mathcal{M}'$ constructed in this section and $\mathcal{M}$ from the previous section is that the new MC $\mathcal{M}'$ needs to satisfy that $\mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}'}(s_0) = 1$ (s_0 is the initial state of $\mathcal{M}'$), while still forcing the candidate NBW $\mathcal{C}$ to make decisions that lose probability of success, leading to $\mathrm{PSyn}_{\mathcal{C}}^{\mathcal{M}'}(s_0) < 1$. This makes the construction of $\mathcal{M}'$ more complex, but establishes that qualitative and full GFM are equivalent properties.

The MC will also be constructed by parts and it has a central part. It will also have the MCs $\mathcal{M}_q$ for each non QGFM state q and $\mathcal{M}_t$ for each non-residual transition t from the previous section. We now describe the three potential problems of $\mathcal{M}$ of the previous section and the possible remedies.

The first potential problem is in the central part as it might contain prefixes that cannot be extended to words in $\mathcal{L}(\mathcal{C})$. Such prefixes should be excluded. This can be addressed by building a cross product with a deterministic safety automaton that recognises all the prefixes of $\mathcal{L}(\mathcal{C})$ (the safety hull of the language of $\mathcal{C}$) and removing the states that have no outgoing transitions in the product.

The second problem is caused by the transitions to all MCs $\mathcal{M}_q$ and $\mathcal{M}_t$ from every state in the central part. This can, however, be avoided by including another safety automaton in the product that tracks, which of these transitions *could* be used by our candidate NBW $\mathcal{C}$ at the moment, and only using those transitions. Note that this retains all transitions used in the proof to establish a difference between $\mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}'}(s_0)$ and $\mathrm{PSyn}_{\mathcal{C}}^{\mathcal{M}'}(s_0)$ while guaranteeing that the semantic probability of winning after progressing to $\mathcal{M}_q$ or $\mathcal{M}_t$ is 1.

Removing the transitions to some of the $\mathcal{M}_q$ and $\mathcal{M}_t$ can potentially create a third problem, namely that no transitions to $\mathcal{M}_q$ and $\mathcal{M}_t$ are left so that there is no way leaving the central part of the MC. To address this problem, we can create a new MC for each state in the central part so that a word starting from the initial state of the MC can always be extended to an accepting word in $\mathcal{L}(\mathcal{C})$ by transitioning to this new MC. How such MCs can be constructed is detailed later.

Zooming in on the construction, the MC $\mathcal{M}$ should satisfy that all the finite runs that start from an initial state s_0, before transitioning to $\mathcal{M}_q$ or $\mathcal{M}_t$, can be extended to a word in the language of $\mathcal{C}$ are retained. The language of all such initial sequences is a safety language, and it is easy to construct an automaton that (1) recognises this safety language and (2) retains the knowledge of how to complete each word in the language of $\mathcal{C}$. To create this automaton, we first determinise $\mathcal{C}$ to a deterministic parity word automaton $\mathcal{D}$ [13]. We then remove all non-productive states from $\mathcal{D}$ and mark all states final, yielding the safety automaton[7] $\mathcal{D}'$.

The two automata $\mathcal{D}'$ and $\mathcal{D}$ can be used to address all the problems we have identified. To address the first problem, we build a cross product MC of $\mathcal{M}_\Sigma$ (the central part of $\mathcal{M}$ in last section) and $\mathcal{D}'$. We then remove all the states in the product MC without any outgoing transitions and make the resulting MC the new central part denoted by $\mathcal{M}'_\Sigma$. Every state in $\mathcal{M}'_\Sigma$ is of the form (s_σ, p) where $s_\sigma \in S_\Sigma$ is from $\mathcal{M}_\Sigma$ and $p \in \mathcal{D}'$.

The states of the deterministic automaton $\mathcal{D}$ (and thus those of $\mathcal{D}'$) also provide information about the possible states of $\mathcal{C}$ that could be after the prefix we have seen so far. To address the second problem, we use this information to overestimate whether $\mathcal{C}$ could be in some state q, or use a transition t, which in turn is good enough for deciding whether or not to transition to the initial states of $\mathcal{M}_q$ resp. $\mathcal{M}_t$ from every state of the new central part.

To address the third problem, we build, for every state p of $\mathcal{D}'$ and every letter $\sigma \in \Sigma$ such that σ can be extended to an accepted word from state p, an MC $\mathcal{M}_{p,\sigma}$ that produces a single ω-regular word (sometimes referred to as lasso word) $w_{p,\sigma}$ with probability 1. The word $\sigma \cdot w_{p,\sigma}$ will be accepted from state p (or: by $\mathcal{D}_p$). From every state (s_σ, p) of the central part, there is a transition to the initial state of $\mathcal{M}_{p,\sigma}$.

[7] From every state p in $\mathcal{D}'$, we can construct an extension to an accepted word by picking an accepted lasso path through $\mathcal{D}$ that starts from p. Note that $\mathcal{D}'$ is not complete, but every state has some successor.

The final MC transitions uniformly at random, from a state (s_σ, p) in $\mathcal{M}'_\Sigma$, to one of its successor states, which comprise the initial state of $\mathcal{M}_{p,\sigma}$ and the initial states of some of the individual MCs $\mathcal{M}_q$ and $\mathcal{M}_t$.

▶ **Lemma 16.** *If $\mathcal{L}(\mathcal{S}) \neq \mathcal{L}(\mathcal{T})$ or $\mathcal{S}$ is not GFG, the candidate NBW $\mathcal{C}$ is not QGFM.*

Proof. The proof of the difference in the probability of winning in case $\mathcal{L}(\mathcal{S}) \neq \mathcal{L}(\mathcal{T})$ or in case $\mathcal{S}$ is not GFG are the same as in Lemma 15.

We additionally have to show that $\mathrm{PSem}_{\mathcal{C}}^{\mathcal{M}'}(s_0) = 1$. But this is easily provided by the construction: when we move on to some $\mathcal{M}_{p,\sigma}$, $\mathcal{M}_q$, or $\mathcal{M}_t$, we have sure, almost sure, and sure satisfaction, respectively, of the property by construction, while staying for ever in the central part of the new MC happens with probability 0. ◀

By definition, if a candidate NBW $\mathcal{C}$ is GFM, it is QGFM. Together with Lemma 14 and Lemma 16, we have that $\mathcal{L}(\mathcal{S}) = \mathcal{L}(\mathcal{T})$ and $\mathcal{S}$ is GFG iff the candidate NBW $\mathcal{C}$ is QGFM. Thus, we have

▶ **Theorem 17.** *The candidate NBW $\mathcal{C}$ is GFM if, and only if, $\mathcal{C}$ is QGFM.*

By Theorem 13 and Theorem 17, we have:

▶ **Corollary 18.** *The problem of whether or not a given NBW is GFM is in* EXPTIME.

Likewise, by Theorem 9, Theorem 10, and Theorem 17, we have:

▶ **Corollary 19.** *The problem of whether or not a given NBW is QGFM is* PSPACE*-hard. Given a QGFM NBW and a bound k, the problem whether there is a language equivalent QGFM NBW with at most k states is* PSPACE*-hard. It is* PSPACE*-hard even for (fixed) $k = 2$.*

6 Discussion

We have started out with introducing the prima facie simpler auxiliary concept of *qualitative* GFM-ness.

We have then established that deciding GFM-ness is PSPACE-hard by a reduction from the NFA universality problem and developed an algorithm for checking *qualitative* GFM-ness in EXPTIME.

We then closed with first characterising GFM-ness with a heavy use of QGFM-ness tests, only to find that this characterisation also proves to be a necessary requirement for QGFM-ness, which led to a collapse of the qualitative and full GFM-ness. The hardness results for GFM-ness therefore carry over to QGFM-ness, while the decision procedure for QGFM-ness proves to be a decision procedure for GFM-ness by itself.

References

1 Luca Alfaro. *Formal Verification of Probabilistic Systems.* PhD thesis, Stanford University, Stanford, CA, USA, 1998.

2 Marc Bagnol and Denis Kuperberg. Büchi good-for-games automata are efficiently recognizable. In *FSTTCS*, 2018.

3 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, 2008.

4 Andrea Bianco and Luca de Alfaro. Model checking of probabalistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science, 15th Conference, Bangalore, India, December 18-20, 1995, Proceedings*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 1995.

5 Thomas Colcombet. Forms of determinism for automata (invited talk). In Christoph Dürr and Thomas Wilke, editors, *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th – March 3rd, 2012, Paris, France*, volume 14 of *LIPIcs*, pages 1–23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.

6 Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–907, July 1995.

7 Ernst Moritz Hahn, Guangyuan Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. Lazy probabilistic model checking without determinisation. In *Concurrency Theory*, pages 354–367, 2015.

8 Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Omega-regular objectives in model-free reinforcement learning. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 395–412, Cham, 2019. Springer International Publishing.

9 Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Good-for-MDPs automata for probabilistic analysis and reinforcement learning. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 306–323, Cham, 2020. Springer International Publishing.

10 Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In *Computer Science Logic*, pages 394–409, September 2006. LNCS 4207.

11 Denis Kuperberg and Michał Skrzypczak. On determinisation of good-for-games automata. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 299–310, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

12 Dominique Perrin and Jean-Éric Pin. *Infinite Words: Automata, Semigroups, Logic and Games.* Elsevier, 2004.

13 Nir Piterman. From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata. *Log. Methods Comput. Sci.*, 3(3), 2007.

14 Amir Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

15 Sven Schewe. Synthesis for probabilistic environments. In *4th International Symposium on Automated Technology for Verification and Analysis (ATVA 2006)*, pages 245–259. Springer Verlag, 2006.

16 Sven Schewe, Qiyi Tang, and Tansholpan Zhanabekova. Deciding what is good-for-mdps, 2023. arXiv:2202.07629.

17 Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínský. Limit-deterministic Büchi automata for linear temporal logic. In *Computer Aided Verification*, pages 312–332, 2016. LNCS 9780.

18 Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 – May 2, 1973, Austin, Texas, USA*, pages 1–9. ACM, 1973.

19 Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, second edition, 2018.

20 Wolfgang Thomas. *Handbook of Theoretical Computer Science*, chapter Automata on Infinite Objects, pages 133–191. The MIT Press/Elsevier, 1990.

21 Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Foundations of Computer Science*, pages 327–338, 1985.

The Semilinear Home-Space Problem Is Ackermann-Complete for Petri Nets

Petr Jančar

Dept of Comp. Sci., Faculty of Science, Palacký University Olomouc, Czech Republic

Jérôme Leroux

LaBRI, CNRS, Univ. Bordeaux, France

Abstract

A set of configurations H is a home-space for a set of configurations X of a Petri net if every configuration reachable from (any configuration in) X can reach (some configuration in) H. The semilinear home-space problem for Petri nets asks, given a Petri net and semilinear sets of configurations X, H, if H is a home-space for X. In 1989, David de Frutos Escrig and Colette Johnen proved that the problem is decidable when X is a singleton and H is a finite union of linear sets with the same periods. In this paper, we show that the general (semilinear) problem is decidable. This result is obtained by proving a duality between the reachability problem and the non-home-space problem. In particular, we prove that for any Petri net and any linear set of configurations L we can effectively compute a semilinear set C of configurations, called a non-reachability core for L, such that for every set X the set L is not a home-space for X if, and only if, C is reachable from X. We show that the established relation to the reachability problem yields the Ackermann-completeness of the (semilinear) home-space problem. For this we also show that, given a Petri net with an initial marking, the set of minimal reachable markings can be constructed in Ackermannian time.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Petri nets, home-space property, semilinear sets, Ackermannian complexity

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.36

Funding *Jérôme Leroux*: supported by the grant ANR-17-CE40-0028 of the French National Research Agency ANR (project BRAVAS).

1 Introduction

Petri nets provide a popular formal method for modelling and analysing parallel processes. The standard model is not Turing-complete, and thus many analyzed properties are decidable; we can refer to [6] as to one of the first survey papers on this issue.

A central algorithmic problem for Petri nets is reachability: given a Petri net A and two configurations $\mathbf{x}$ and $\mathbf{y}$, decide whether there exists an execution of A from $\mathbf{x}$ to $\mathbf{y}$. In fact, many important computational problems in logic and complexity reduce or are even equivalent to this problem (we can refer, e.g., to [19, 9] to exemplify this). It was nontrivial to show that the reachability problem is decidable [16], and recently the complexity of this problem was proved to be extremely high, namely Ackermann-complete (see [15] for the upper-bound and [3, 14, 4] for the lower-bound).

The reachability problem for Petri nets can be generalized to semilinear sets, a class of geometrical sets that coincides with the sets definable in Presburger arithmetic [8]. The semilinear reachability problem for Petri nets asks, given a Petri net A and (presentations of) semilinear sets of configurations $\mathbf{X}$,$\mathbf{Y}$, if there exists an execution from a configuration in $\mathbf{X}$ to a configuration in $\mathbf{Y}$. Denoting by $\textsc{post}_A^*(\mathbf{X})$ the set of configurations reachable from $\mathbf{X}$ and by $\textsc{pre}_A^*(\mathbf{Y})$ the set of configurations that can reach a configuration in $\mathbf{Y}$, the semilinear reachability problem thus asks, in fact, if the intersection $\textsc{post}_A^*(\mathbf{X}) \cap \textsc{pre}_A^*(\mathbf{Y})$ is nonempty. This problem can be easily reduced to the classical reachability problem for Petri nets (where $\mathbf{X}$ and $\mathbf{Y}$ are singletons).

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 36; pp. 36:1–36:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The semilinear home-space problem is a problem that seems to be similar to the semilinear reachability problem at a first sight. This problem asks, given a Petri net A, and two semilinear sets $\mathbf{X}, \mathbf{H}$, if every configuration reachable from $\mathbf{X}$ can reach $\mathbf{H}$, hence if $\textsc{post}^*_A(\mathbf{X}) \subseteq \textsc{pre}^*_A(\mathbf{H})$. In 1989, David de Frutos Escrig and Colette Johnen [5] proved that the semilinear home-space problem is decidable for instances where $\mathbf{X}$ is a singleton set and $\mathbf{H}$ is a finite union of linear sets using the same periods; they left the general case open. In fact, the general problem seems close to the decidability/undecidability border, since the reachability set inclusion problem, which can be viewed as asking if $\textsc{post}^*_A(\mathbf{x}) \subseteq \textsc{pre}^*_B(\mathbf{y})$ where A, B are Petri nets of the same dimension (i.e., with the same sets of places), is undecidable [1, 10], even when the dimension of A, B is fixed to a small value [11].

Our contribution. In this paper, we show that the general semilinear home-space problem is decidable. This result is obtained by proving a duality between the reachability problem and the non-home-space problem. A crucial point consists in proving that for any Petri net A and for any linear set of configurations $\mathbf{L}$, we can effectively compute a semilinear "non-reachability core" $\mathbf{C}$ such that for every set $\mathbf{X}$ the set $\mathbf{L}$ is not a home-space for $\mathbf{X}$ if, and only if, $\mathbf{C}$ is reachable from $\mathbf{X}$. By a technical analysis using the known complexity results for reachability we show that the (semilinear) home-space problem is Ackermann-complete. As an ingredient, we also show that, given a Petri net with an initial marking, the set of minimal reachable markings can be constructed in Ackermannian time.

Organization of the paper. Section 2 states our theorems, after some preliminaries. Section 3 shows the hardness results, and Sections 4 and 5 give a decidability proof. Sections 6 and 7 contain the complexity analysis, and Section 8 adds some concluding remarks.

2 Basic Notions, and Main Results

In this section, we introduce basic notions and notation, and state the main results.

Notation for Vectors of Nonnegative Integers

By $\mathbb{N}$ we denote the set $\{0, 1, 2, \dots\}$ of nonnegative integers. For $i, j \in \mathbb{N}$, by $[i, j]$ we denote the set $\{i, i{+}1, \dots, j\}$.

For (a dimension) $d \in \mathbb{N}$, the elements of $\mathbb{N}^d$ are called (d-dimensional) *vectors*; they are denoted in bold face, and for $\mathbf{x} \in \mathbb{N}^d$ we put $\mathbf{x} = (\mathbf{x}(1), \mathbf{x}(2), \dots, \mathbf{x}(d))$ so that we can refer to the vector components. We use the component-wise sum $\mathbf{x} + \mathbf{y}$ of vectors, and their component-wise order $\mathbf{x} \leq \mathbf{y}$. For $c \in \mathbb{N}$, we put $c \cdot \mathbf{x} = (c \cdot \mathbf{x}(1), c \cdot \mathbf{x}(2), \dots, c \cdot \mathbf{x}(d))$. By the *norm of* $\mathbf{x}$, denoted $\|\mathbf{x}\|$, we mean the sum of components, i.e., $\|\mathbf{x}\| = \sum_{i=1}^{d} \mathbf{x}(i)$.

By $\mathbf{0}$ we denote the zero vector whose dimension is always clear from its context. Occasionally we slightly abuse notation by presenting a vector as a mix of subvectors and integers; in particular, given $\mathbf{x} \in \mathbb{N}^d$ and $y_1, y_2, \dots, y_m \in \mathbb{N}$, we might write $(\mathbf{x}, y_1, y_2, \dots, y_m)$ to denote the $(d{+}m)$-dimensional vector $(\mathbf{x}(1), \mathbf{x}(2), \dots, \mathbf{x}(d), y_1, y_2, \dots, y_m)$.

Given a set $\mathbf{X} \subseteq \mathbb{N}^d$, by $\overline{\mathbf{X}}$ we denote its complement, i.e., $\overline{\mathbf{X}} = \mathbb{N}^d \setminus \mathbf{X}$.

Linear and Semilinear Sets of Vectors

A *set* $\mathbf{L} \subseteq \mathbb{N}^d$ is *linear* if there are d-dimensional vectors $\mathbf{b}$, the *basis*, and $\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_k$, the *periods* (for $k \in \mathbb{N}$), such that $\mathbf{L} = \{\mathbf{x} \in \mathbb{N}^d \mid \mathbf{x} = \mathbf{b} + \mathbf{u}(1) \cdot \mathbf{p}_1 + \mathbf{u}(2) \cdot \mathbf{p}_2 \cdots + \mathbf{u}(k) \cdot \mathbf{p}_k$ for some $\mathbf{u} \in \mathbb{N}^k\}$. In this case, by a *presentation of* $\mathbf{L}$ we mean the tuple $(\mathbf{b}, \mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_k)$.

A *set* $\mathbf{S} \subseteq \mathbb{N}^d$ is *semilinear* if it is a finite union of linear sets, i.e. $\mathbf{S} = \mathbf{L}_1 \cup \mathbf{L}_2 \cdots \cup \mathbf{L}_m$ where $\mathbf{L}_i$ are linear sets (for all $i \in [1, m]$). In this case, by a *presentation of* $\mathbf{S}$ we mean the sequence of presentations of $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_m$. When we say that a *semilinear set* $\mathbf{S}$ is given, we mean that we are given a presentation of $\mathbf{S}$; when we say that $\mathbf{S}$ is *effectively constructible* in some context, we mean that there is an algorithm computing its presentation (in the respective context).

We recall that a set $\mathbf{S} \subseteq \mathbb{N}^d$ is semilinear if, and only if, it is expressible in Presburger arithmetic [8]; the respective transformations between presentations and formulas are effective. Hence if $\mathbf{S} \subseteq \mathbb{N}^d$ is semilinear, then also its complement $\overline{\mathbf{S}}$ is semilinear, and $\overline{\mathbf{S}}$ is effectively constructible when (a presentation of) $\mathbf{S}$ is given.

Petri Nets

We use a concise definition of (unmarked place/transition) Petri nets. By a *d-dimensional Petri-net action* we mean a pair $a = (\mathbf{a}_-, \mathbf{a}_+) \in \mathbb{N}^d \times \mathbb{N}^d$. With $a = (\mathbf{a}_-, \mathbf{a}_+)$ we associate the binary relation $\xrightarrow{a}$ on the set $\mathbb{N}^d$ by putting $(\mathbf{x}+\mathbf{a}_-) \xrightarrow{a} (\mathbf{x}+\mathbf{a}_+)$ for all $\mathbf{x} \in \mathbb{N}^d$. The relations $\xrightarrow{a}$ are naturally extended to the relations $\xrightarrow{\sigma}$ for finite sequences σ of (d-dimensional Petri net) actions.

A *Petri net A of dimension d* (with d places in more traditional definitions) is a finite set of d-dimensional Petri-net actions (transitions). Here the vectors $\mathbf{x} \in \mathbb{N}^d$ are also called *configurations* (markings). On the set $\mathbb{N}^d$ of configurations we define the *reachability relation* $\xrightarrow{A^*}$: we put $\mathbf{x} \xrightarrow{A^*} \mathbf{y}$ if there is $\sigma \in A^*$ such that $\mathbf{x} \xrightarrow{\sigma} \mathbf{y}$. For $\mathbf{x} \in \mathbb{N}^d$ and $\mathbf{X} \subseteq \mathbb{N}^d$ we put $\textsc{post}^*_A(\mathbf{x}) = \{\mathbf{y} \in \mathbb{N}^d \mid \mathbf{x} \xrightarrow{A^*} \mathbf{y}\}$, and $\textsc{post}^*_A(\mathbf{X}) = \bigcup_{\mathbf{x}\in\mathbf{X}} \textsc{post}^*_A(\mathbf{x})$. Symmetrically, for $\mathbf{y} \in \mathbb{N}^d$ and $\mathbf{Y} \subseteq \mathbb{N}^d$ we put $\textsc{pre}^*_A(\mathbf{y}) = \{\mathbf{x} \in \mathbb{N}^d \mid \mathbf{x} \xrightarrow{A^*} \mathbf{y}\}$ and $\textsc{pre}^*_A(\mathbf{Y}) = \bigcup_{\mathbf{y}\in\mathbf{Y}} \textsc{pre}^*_A(\mathbf{y})$. By $\mathbf{X} \xrightarrow{A^*} \mathbf{Y}$ we denote that $\mathbf{x} \xrightarrow{A^*} \mathbf{y}$ for some $\mathbf{x} \in \mathbf{X}$ and $\mathbf{y} \in \mathbf{Y}$, i.e. that $\textsc{post}^*_A(\mathbf{X}) \cap \mathbf{Y} \neq \emptyset$, or equivalently $\mathbf{X} \cap \textsc{pre}^*_A(\mathbf{Y}) \neq \emptyset$.

(Semilinear) Reachability Problem

By the (semilinear) *reachability problem* we mean the following decision problem:

> *Instance:* a d-dimensional Petri net A and presentations of two semilinear sets $\mathbf{X}, \mathbf{Y} \subseteq \mathbb{N}^d$, to which we concisely refer as to the triple $A, \mathbf{X}, \mathbf{Y}$.
>
> *Question:* does $\mathbf{X} \xrightarrow{A^*} \mathbf{Y}$ hold?

In the standard definition of the reachability problem the sets $\mathbf{X}, \mathbf{Y}$ are singletons; the problem is decidable [16], and it has been recently shown to be Ackermann-complete [15, 14, 4]. It is well-known (and easy to show) that the above more general version (the semilinear reachability problem) is tightly related to the standard version, and has thus the same complexity.

▶ Remark 1. We can sketch this tight relation as follows. If $\mathbf{X}$ and $\mathbf{Y}$ are linear, with presentations $(\mathbf{b}, \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k)$ and $(\mathbf{b}', \mathbf{p}'_1, \mathbf{p}'_2, \ldots, \mathbf{p}'_{k'})$ respectively, then it suffices to ask whether $\mathbf{b} \xrightarrow{(A')^*} \mathbf{b}'$ where A' arises from A by adding the actions $(\mathbf{0}, \mathbf{p}_i)$ for all $i \in [1, k]$ and $(\mathbf{p}'_i, \mathbf{0})$ for all $i \in [1, k']$. Now if $\mathbf{X} = \mathbf{L}_1 \cup \mathbf{L}_2 \cdots \cup \mathbf{L}_m$ and $\mathbf{Y} = \mathbf{L}'_1 \cup \mathbf{L}'_2 \cdots \cup \mathbf{L}'_{m'}$, then it suffices to check if $\mathbf{L}_i \xrightarrow{A^*} \mathbf{L}'_j$ for some $i \in [1, m]$ and $j \in [1, m']$. (In fact, there is a polynomial reduction of the general version to the standard one, which increases the dimension and uses the added vector components to mimic control states.)

Semilinear Home-Space Problem

For a Petri net A of dimension d and two sets $\mathbf{X}, \mathbf{H} \subseteq \mathbb{N}^d$, we call $\mathbf{H}$ a *home-space for* $(A, \mathbf{X})$, or just for $\mathbf{X}$ when A is clear from context, if $\textsc{post}^*_A(\mathbf{X}) \subseteq \textsc{pre}^*_A(\mathbf{H})$. (A trivial observation is that $\textsc{post}^*_A(\mathbf{X})$ is a home-space for $\mathbf{X}$.)

We note that the above (semilinear) reachability problem in fact asks, given $A, \mathbf{X}, \mathbf{Y}$, if $\textsc{post}^*_A(\mathbf{X}) \cap \textsc{pre}^*_A(\mathbf{Y}) \neq \emptyset$. The *semilinear home-space problem* is defined as follows:

> *Instance:* a triple $A, \mathbf{X}, \mathbf{H}$ where A is a Petri net, of dimension d, and $\mathbf{X}$, $\mathbf{H}$ are two (finitely presented) semilinear subsets of $\mathbb{N}^d$.
> *Question:* is $\textsc{post}^*_A(\mathbf{X}) \subseteq \textsc{pre}^*_A(\mathbf{H})$ (i.e., is $\mathbf{H}$ a home-space for $\mathbf{X}$) ?

Our main result is stated by Theorem 3. Nevertheless, we first prove the weaker claim, Theorem 2, that answers an open question from [5] and does not need the technicalities related to the complexity analysis.

▶ **Theorem 2.** *The semilinear home-space problem is decidable.*

▶ **Theorem 3.** *The semilinear home-space problem is Ackermann-complete.*

We remark that by [5] we know that the home-space problem is decidable for the instances A, $\mathbf{X}$, $\mathbf{H}$ where $\mathbf{X}$ is a singleton set, and $\mathbf{H}$ is a finite union of linear sets with the same periods; this was established by a Turing reduction to the reachability problem. The decidability in the case where $\mathbf{H}$ is a general semilinear set was left open in [5]; this more general problem indeed looks more subtle but we manage to provide a solution here. Before doing this, we note in Section 3 that the problem has also a high computational complexity, and can be naturally viewed as residing at the decidability/undecidability border.

3 Home-Space Problem is Hard

We first note that even a simple version of the home-space problem is at least as hard as (non)reachability, and thus Ackermann-hard. We use a polynomial reduction that increases the Petri net dimension, by additional vector components that can be viewed as control states. (It would be natural to use the model of *vector addition systems with states* but we do not introduce them formally in this paper.)

▶ **Proposition 4.** *The non-reachability problem is polynomially reducible to the home-space problem restricted to the instances* $A, \mathbf{X}, \mathbf{H}$ *where* $\mathbf{X}$ *and* $\mathbf{H}$ *are singletons.*

Proof. Let us consider a Petri net A of dimension d and two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{N}^d$, as an instance of the (non)reachability problem. We create the $(d{+}3)$-dimensional Petri net A' so that each action $a = (\mathbf{a}_-, \mathbf{a}_+)$ of A is transformed to the action $a' = ((\mathbf{a}_-, 1, 0, 0), (\mathbf{a}_+, 1, 0, 0))$ of A', and A' has also the additional actions $((\mathbf{y}, 1, 0, 0), (\mathbf{0}, 0, 1, 0))$, $((\mathbf{0}, 1, 0, 0), (\mathbf{0}, 0, 0, 1))$, and the actions $((\mathbf{i}_j, 0, 1, 0), (\mathbf{0}, 0, 0, 1))$, $((\mathbf{i}_j, 0, 0, 1), (\mathbf{0}, 0, 0, 1))$ for all $j \in [1, d]$, where $\mathbf{i}_j \in \mathbb{N}^d$ satisfies $\mathbf{i}_j(j) = 1$ and $\mathbf{i}_j(i) = 0$ for all $i \neq j$.

We verify that $\mathbf{x} \xrightarrow{A^*} \mathbf{y}$ if, and only if, $\{(\mathbf{0}, 0, 0, 1)\}$ is not a home-space for $(A', \{(\mathbf{x}, 1, 0, 0)\})$:

- if $\mathbf{x} \xrightarrow{A^*} \mathbf{y}$, then $(\mathbf{x}, 1, 0, 0) \xrightarrow{(A')^*} (\mathbf{y}, 1, 0, 0) \xrightarrow{(A')^*} (\mathbf{0}, 0, 1, 0)$, and $(\mathbf{0}, 0, 0, 1)$ is not reachable from $(\mathbf{0}, 0, 1, 0)$;
- if $\mathbf{x} \stackrel{A^*}{\not\longrightarrow} \mathbf{y}$, then any configuration reachable from $(\mathbf{x}, 1, 0, 0)$ in A' is in one of the forms $(\mathbf{y}', 1, 0, 0)$, $(\mathbf{z}, 0, 1, 0)$, $(\mathbf{z}', 0, 0, 1)$ where $\mathbf{y}' \neq \mathbf{y}$ and $\mathbf{z} \neq \mathbf{0}$, and $(\mathbf{0}, 0, 0, 1)$ is clearly reachable from all of them. ◀

Now we note that a slight generalization of the semilinear home-space problem is undecidable; it is the case when instead of semilinear sets $\mathbf{H}$ in the instances $A, \mathbf{X}, \mathbf{H}$ we allow $\mathbf{H}$ to be reachability sets of Petri nets (that are a special case of so called *almost semilinear sets* [13]).

▶ **Proposition 5.** *Given Petri nets A, B of the same dimension d, and two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{N}^d$, it is undecidable if $\text{POST}^*_B(\mathbf{y})$ is a home-space for $(A, \{\mathbf{x}\})$.*

Proof. We recall that the reachability set inclusion problem is undecidable for Petri nets (and for the equivalent model of vector addition systems); see [1, 10, 11]. Hence it is undecidable, given Petri nets A, B of the same dimension d and $\mathbf{x}, \mathbf{y} \in \mathbb{N}^d$, whether $\text{POST}^*_A(\mathbf{x}) \subseteq \text{POST}^*_B(\mathbf{y})$. If A' arises from A by replacing each action $a = (\mathbf{a}_-, \mathbf{a}_+)$ with $a' = ((\mathbf{a}_-, 1), (\mathbf{a}_+, 1))$ and by adding the action $((\mathbf{0}, 1), (\mathbf{0}, 0))$, and B' arises from B by replacing each $b = (\mathbf{b}_-, \mathbf{b}_+)$ with $b' = ((\mathbf{b}_-, 0), (\mathbf{b}_+, 0))$, then we obviously have that $\text{POST}^*_{B'}((\mathbf{y}, 0))$ is a home-space for $(A', (\mathbf{x}, 1))$ if, and only if, $\text{POST}^*_A(\mathbf{x}) \subseteq \text{POST}^*_B(\mathbf{y})$. ◀

▶ Remark 6. Since [11] shows, in fact, that the reachability set inclusion (or equality) problem is undecidable even for some fixed five-dimensional vector addition systems with states (VASSs), we could appropriately strengthen Proposition 5; but we do not pursue this technical issue here.

We can note that the undecidability of the question if $\text{POST}^*_B(\mathbf{x}) \subseteq \text{POST}^*_A(\mathbf{y})$ entails that the question if $\text{POST}^*_B(\mathbf{x}) \subseteq \text{PRE}^*_A(\mathbf{y})$ is also undecidable (since $\text{POST}^*_A(\mathbf{y})$ is equal to $\text{PRE}^*_{A_{rev}}(\mathbf{y})$ where A_{rev} arises from A by reversing each action $(\mathbf{a}_-, \mathbf{a}_+)$ to $(\mathbf{a}_+, \mathbf{a}_-)$). On the other hand, in the next sections we show that the question if $\text{POST}^*_A(\mathbf{x}) \subseteq \text{PRE}^*_A(\mathbf{y})$ is decidable. We will show that, given a d-dimensional Petri net A and $\mathbf{y} \in \mathbb{N}^d$, we can effectively construct a semilinear set (a "non-reachability core") $C \subseteq \mathbb{N}^d$ such that $\text{POST}^*_A(\mathbf{x}) \not\subseteq \text{PRE}^*_A(\mathbf{y})$ if, and only if, $\text{POST}^*_A(\mathbf{x})$ intersects C. The equality of the nets on both sides is crucial, since if $\text{POST}^*_B(\mathbf{x})$ does not intersect C, then this does not entail $\text{POST}^*_B(\mathbf{x}) \subseteq \text{PRE}^*_A(\mathbf{y})$.

4 Decidability of Home-Space via Semilinear Non-Reachability Cores

Now we start to discuss how to decide the semilinear home-space problem. We assume a fixed Petri net A of dimension d if not said otherwise.

We first note that the home-space property can be naturally formulated in terms of "reachability of non-reachability". To this aim we introduce a technical notion and make a related observation.

Given a set $\mathbf{H} \subseteq \mathbb{N}^d$, we say that a set $\mathbf{C} \subseteq \mathbb{N}^d$ is a *non-reachability core for* $\mathbf{H}$ if

1. $\mathbf{C} \not\xrightarrow{A^*} \mathbf{H}$ (hence $\mathbf{C} \subseteq \overline{\text{PRE}^*_A(\mathbf{H})}$ where $\overline{\text{PRE}^*_A(\mathbf{H})} = \mathbb{N}^d \smallsetminus \text{PRE}^*_A(\mathbf{H})$), and
2. for each $\mathbf{x} \in \mathbb{N}^d$, if $\mathbf{x} \not\xrightarrow{A^*} \mathbf{H}$ then $\mathbf{x} \xrightarrow{A^*} \mathbf{C}$ (hence $\overline{\text{PRE}^*_A(\mathbf{H})} \subseteq \text{PRE}^*_A(\mathbf{C})$).

▶ **Proposition 7.** *If $\mathbf{C}$ is a non-reachability core for $\mathbf{H}$, then for every $\mathbf{X} \subseteq \mathbb{N}^d$ we have that*

$\mathbf{H}$ is not a home-space for $\mathbf{X}$ if, and only if, $\mathbf{X} \xrightarrow{A^*} \mathbf{C}$.

Proof. If $\mathbf{X} \xrightarrow{A^*} \mathbf{C}$, then $\mathbf{x} \xrightarrow{A^*} \mathbf{c}$ for some $\mathbf{x} \in \mathbf{X}$ and $\mathbf{c} \in \mathbf{C}$; since $\mathbf{c} \not\xrightarrow{A^*} \mathbf{H}$ by condition 1, $\mathbf{H}$ is not a home-space for $\mathbf{X}$.

If $\mathbf{H}$ is not a home-space for $\mathbf{X}$, then we have $\mathbf{x} \xrightarrow{A^*} \mathbf{x}' \not\xrightarrow{A^*} \mathbf{H}$ for some $\mathbf{x} \in \mathbf{X}$ and some $\mathbf{x}'$. By condition 2, $\mathbf{x}' \xrightarrow{A^*} \mathbf{C}$; hence $\mathbf{x} \xrightarrow{A^*} \mathbf{C}$, which entails $\mathbf{X} \xrightarrow{A^*} \mathbf{C}$. ◀

We note that $\overline{\text{PRE}^*_A(\mathbf{H})}$ is clearly a non-reachability core for $\mathbf{H}$, and the question whether $\mathbf{X} \xrightarrow{A^*} \overline{\text{PRE}^*_A(\mathbf{H})}$ asks, in fact, whether $\mathbf{H}$ is not a home-space for $\mathbf{X}$. To decide the question whether $\mathbf{X} \xrightarrow{A^*} \overline{\text{PRE}^*_A(\mathbf{H})}$ when $\mathbf{H}$ is a semilinear set (given by a standard presentation), a natural idea is to look if there is an effectively constructible semilinear non-reachability core $\mathbf{C}$ for $\mathbf{H}$ (where $\mathbf{C} \subseteq \overline{\text{PRE}^*_A(\mathbf{H})}$); we recall that the question whether $\mathbf{X} \xrightarrow{A^*} \mathbf{C}$ is decidable for semilinear sets $\mathbf{X}, \mathbf{C}$. In fact, we manage to realize this idea directly only in the case when $\mathbf{H}$ is a linear set:

▶ **Lemma 8.** *Given a Petri net A of dimension d, and (a presentation of) a linear set $\mathbf{L} \subseteq \mathbb{N}^d$, there is an effectively constructible semilinear non-reachability core $\mathbf{C}$ for $\mathbf{L}$.*

This crucial lemma will be proved in the next section (Section 5). Here we show the decidability of the semilinear home-space problem when assuming the lemma. To this aim we first note a useful fact captured by the following proposition.

▶ **Proposition 9.** *Assuming a Petri net A of dimension d, if $\mathbf{H} = \mathbf{H}_1 \cup \mathbf{H}_2 \cdots \cup \mathbf{H}_m$ and $\mathbf{C}_1, \mathbf{C}_2, \dots, \mathbf{C}_m$ are such that $\mathbf{C}_i$ is a non-reachability core for $\mathbf{H}_i$ for each $i \in [1, m]$, then for each $\mathbf{X} \subseteq \mathbb{N}^d$ we have that $\mathbf{H}$ is not a home-space for $\mathbf{X}$ if, and only if, there is an execution*

$$\mathbf{x}_0 \xrightarrow{A^*} \mathbf{x}_1 \xrightarrow{A^*} \mathbf{x}_2 \cdots \xrightarrow{A^*} \mathbf{x}_m \tag{1}$$

where $\mathbf{x}_0 \in \mathbf{X}$, and $\mathbf{x}_i \in \mathbf{C}_i$ for each $i \in [1, m]$. (We do not exclude the cases $\mathbf{x}_i = \mathbf{x}_{i+1}$.)

Proof. Given an execution (1), the facts that $\mathbf{x}_i \in \mathbf{C}_i$ and $\mathbf{C}_i$ is a non-reachability core for $\mathbf{H}_i$ (hence $\mathbf{C}_i \subseteq \overline{\text{PRE}^*_A(\mathbf{H}_i)}$) entail $\mathbf{x}_i \not\xrightarrow{A^*} \mathbf{H}_i$, for all $i \in [1, m]$. The facts $\mathbf{x}_i \not\xrightarrow{A^*} \mathbf{H}_i$ and $\mathbf{x}_i \xrightarrow{A^*} \mathbf{x}_m$ entail that $\mathbf{x}_m \not\xrightarrow{A^*} \mathbf{H}_i$ (for all $i \in [1, m]$). Hence $\mathbf{x}_m \not\xrightarrow{A^*} \mathbf{H}$ (where $\mathbf{H} = \mathbf{H}_1 \cup \mathbf{H}_2 \cdots \cup \mathbf{H}_m$), and the facts $\mathbf{x}_0 \in \mathbf{X}$ and $\mathbf{x}_0 \xrightarrow{A^*} \mathbf{x}_m \not\xrightarrow{A^*} \mathbf{H}$ entail that $\mathbf{H}$ is not a home-space for $\mathbf{X}$.

Conversely, we assume a set $\mathbf{X} \subseteq \mathbb{N}^d$ for which $\mathbf{H}$ is not a home-space. Hence there exist configurations $\mathbf{x}_0, \mathbf{x}'_0$ such that $\mathbf{x}_0 \in \mathbf{X}$ and $\mathbf{x}_0 \xrightarrow{A^*} \mathbf{x}'_0 \not\xrightarrow{A^*} \mathbf{H}$. In particular $\mathbf{x}'_0 \not\xrightarrow{A^*} \mathbf{H}_1$, and thus $\mathbf{H}_1$ is not a home-space for $\{\mathbf{x}'_0\}$. Since $\mathbf{C}_1$ is a non-reachability core for $\mathbf{H}_1$, we have $\mathbf{x}'_0 \xrightarrow{A^*} \mathbf{x}_1$ for some $\mathbf{x}_1 \in \mathbf{C}_1$. Since $\mathbf{x}'_0 \not\xrightarrow{A^*} \mathbf{H}$ and $\mathbf{x}'_0 \xrightarrow{A^*} \mathbf{x}_1$, we have $\mathbf{x}_1 \not\xrightarrow{A^*} \mathbf{H}$, and in particular $\mathbf{x}_1 \not\xrightarrow{A^*} \mathbf{H}_2$. Since $\mathbf{H}_2$ is not a home-space for $\{\mathbf{x}_1\}$ and $\mathbf{C}_2$ is a non-reachability core for $\mathbf{H}_2$, we get $\mathbf{x}_1 \xrightarrow{A^*} \mathbf{x}_2$ for some $\mathbf{x}_2 \in \mathbf{C}_2$. Continuing in this way, we successively derive the existence of an execution (1). ◀

The next proposition gives us the final ingredient for showing an algorithm deciding the semilinear home-space problem.

▶ **Proposition 10.** *Given a Petri net A of dimension d, and (presentations of) semilinear subsets $\mathbf{X}_0, \mathbf{X}_1, \dots, \mathbf{X}_m$ of $\mathbb{N}^d$, the existence of an execution*

$$\mathbf{x}_0 \xrightarrow{A^*} \mathbf{x}_1 \xrightarrow{A^*} \mathbf{x}_2 \cdots \xrightarrow{A^*} \mathbf{x}_m \tag{2}$$

where $\mathbf{x}_i \in \mathbf{X}_i$ for each $i \in [0, m]$ is decidable (by a reduction to reachability).

Proof. By a standard construction, we can build a Petri net with a bigger dimension and an initial configuration that first generates m copies of some $\mathbf{x}_0 \in \mathbf{X}_0$, then performs an execution of A from $\mathbf{x}_0$ on all these copies, while at some moment it freezes some configuration $\mathbf{x}_1$ reached in the first copy, later it freezes some $\mathbf{x}_2$ reached in the second copy, etc.; at the end it starts a "testing part" that enables to reach the zero configuration if, and only if, $\mathbf{x}_1 \in \mathbf{X}_1$, $\mathbf{x}_2 \in \mathbf{X}_2$, …, $\mathbf{x}_m \in \mathbf{X}_m$. ◀

We note that a proof of Theorem 2 is now clear: Given a Petri net A of dimension d and two semilinear sets $\mathbf{X}, \mathbf{H} \subseteq \mathbb{N}^d$, we use that $\mathbf{H} = \mathbf{H}_1 \cup \mathbf{H}_2 \ldots \cup \mathbf{H}_m$ where $\mathbf{H}_i$ are linear sets, and by Lemma 8 we can construct a semilinear non-reachability core $\mathbf{C}_i$ for $\mathbf{H}_i$, for each $i \in [1, m]$. Then we ask if there is an execution (1) from Proposition 9; this can be decided effectively by Proposition 10 (since (1) is a particular case of (2) in this case).

5 Effective Semilinear Non-Reachability Core for Linear Set

Before proving Lemma 8 in Section 5.2, in Section 5.1 we recall an important ingredient dealing with computing the minimal elements in some sets $\mathbf{X} \subseteq \mathbb{N}^d$; its use in Petri nets originates in the work by Valk and Jantzen [20].

5.1 Computing $\min(\mathbf{X})$ for $\mathbf{X} \subseteq \mathbb{N}^d$

For $\mathbf{X} \subseteq \mathbb{N}^d$ we call a vector $\mathbf{m} \in \mathbf{X}$ *minimal in* $\mathbf{X}$ if there is no vector $\mathbf{x} \in \mathbf{X}$ such that $\mathbf{x} \leq \mathbf{m}$ and $\mathbf{x} \neq \mathbf{m}$. (We recall that $\mathbf{x} \leq \mathbf{y}$ denotes that $\mathbf{x}(i) \leq \mathbf{y}(i)$ for all $i \in [1, d]$.) By $\min(\mathbf{X})$ we denote *the set of minimal elements in* $\mathbf{X}$. Since $\leq$ is a well-partial-order on $\mathbb{N}^d$ (by Dickson's lemma), the set $\min(\mathbf{X})$ is finite and for every $\mathbf{x} \in \mathbf{X}$ there exists (at least one) $\mathbf{m} \in \min(\mathbf{X})$ such that $\mathbf{m} \leq \mathbf{x}$.

As a basis for computing $\min(\mathbf{X})$ (for special sets $\mathbf{X} \subseteq \mathbb{N}^d$), it is useful to extend the ordered set $(\mathbb{N}, \leq)$ with an extra element $\omega \notin \mathbb{N}$ so that $x \leq \omega$ for every $x \in \mathbb{N}_\omega$, where $\mathbb{N}_\omega$ denotes $\mathbb{N} \cup \{\omega\}$. By $\mathbb{N}_\omega^d$ we denote the set of d-dimensional vectors over $\mathbb{N}_\omega$; the (component-wise) order $\leq$ on $\mathbb{N}^d$ is naturally extended to $\mathbb{N}_\omega^d$. For $\mathbf{v} \in \mathbb{N}_\omega^d$ we put $\downarrow\mathbf{v} = \{\mathbf{y} \in \mathbb{N}^d \mid \mathbf{y} \leq \mathbf{v}\}$. Hence even when $\mathbf{v}$ has some ω-components, $\mathbf{y} \in \downarrow\mathbf{v}$ has none.

For $\mathbf{X} \subseteq \mathbb{N}^d$ we trivially have $\min(\mathbf{X}) = \min(\mathbf{X} \cap \downarrow(\omega, \omega, \ldots, \omega))$. If we want to describe $\min(\mathbf{X} \cap \downarrow\mathbf{v})$, for $\mathbf{v} \in \mathbb{N}_\omega^d$, and we have some $\mathbf{y} \in (\mathbf{X} \cap \downarrow\mathbf{v})$, then we observe that

$$\min(\mathbf{X} \cap \downarrow\mathbf{v}) = \min\Big(\{\mathbf{y}\} \cup \min\big(\mathbf{X} \cap (\downarrow\mathbf{v} \smallsetminus \{\mathbf{x} \mid \mathbf{y} \leq \mathbf{x}\})\big)\Big).$$

To write this more concretely, by $\mathbf{v}[i \leftarrow k]$, where $i \in [1, d]$ and $k \in \mathbb{N}$, we denote the vector $\mathbf{v}' \in \mathbb{N}_\omega^d$ coinciding with $\mathbf{v}$ except that we have $\mathbf{v}'(i) = k$, and we put

$$\delta_{\mathbf{y}}(\mathbf{v}) = \{\mathbf{w} \in \mathbb{N}_\omega^d \mid \mathbf{w} = \mathbf{v}[i \leftarrow (\mathbf{y}(i)-1)], i \in [1, d], \mathbf{y}(i) > 0\}.$$

▶ **Observation 11.** *For all* $\mathbf{v} \in \mathbb{N}_\omega^d$ *and* $\mathbf{y} \in \downarrow\mathbf{v}$ *we have:*

1. *Each* $\mathbf{w} \in \delta_{\mathbf{y}}(\mathbf{v})$ *is strictly less than* $\mathbf{v}$ *(i.e.,* $\mathbf{w} \leq \mathbf{v}$ *and* $\mathbf{w} \neq \mathbf{v}$*).*
2. $\downarrow\mathbf{v} \smallsetminus \{\mathbf{x} \mid \mathbf{y} \leq \mathbf{x}\} = \bigcup_{\mathbf{w} \in \delta_{\mathbf{y}}(\mathbf{v})} \downarrow\mathbf{w}$.

▶ **Observation 12.** *For all* $\mathbf{X} \subseteq \mathbb{N}^d$*,* $\mathbf{v} \in \mathbb{N}_\omega^d$*, and* $\mathbf{y} \in (\mathbf{X} \cap \downarrow\mathbf{v})$ *we have:*

$$\min(\mathbf{X} \cap \downarrow\mathbf{v}) = \min\left(\{\mathbf{y}\} \cup \bigcup_{\mathbf{w} \in \delta_{\mathbf{y}}(\mathbf{v})} \min(\mathbf{X} \cap \downarrow\mathbf{w})\right).$$

Since each strictly decreasing sequence $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots$ of vectors in $\mathbb{N}_\omega^d$ is finite, we easily observe that there is an algorithm stated in the next lemma. Its inputs are special algorithms that we call *set-related algorithms*. Each set-related algorithm is related to some set $\mathbf{X} \subseteq \mathbb{N}^d$ (for some $d \in \mathbb{N}$); given $\mathbf{v} \in \mathbb{N}_\omega^d$, the algorithm decides if $(\mathbf{X} \cap \downarrow\mathbf{v})$ is nonempty, and in the positive case returns some $\mathbf{y} \in (\mathbf{X} \cap \downarrow\mathbf{v})$.

▶ **Lemma 13.** *There is an algorithm that, given a set-related algorithm related to* $\mathbf{X} \subseteq \mathbb{N}^d$*, computes the set* $\min(\mathbf{X})$*.*

▶ Remark 14. In fact, the algorithm claimed by the lemma does not require to get a code of a set-related algorithm; it suffices to get (black-box) access to such an algorithm.

5.2 Proof of Lemma 8

Now we prove Lemma 8:

> *Given a Petri net A of dimension d, and (a presentation of) a linear set $\mathbf{L} \subseteq \mathbb{N}^d$, there is an effectively constructible semilinear non-reachability core $\mathbf{C}$ for $\mathbf{L}$.*

We assume a fixed Petri net A of dimension d, and we first prove the claim for the case where $\mathbf{L}$ is a singleton; hence $\mathbf{L} = \{\mathbf{b}\}$ (there is a basis $\mathbf{b} \in \mathbb{N}^d$, but no periods). We observe that if $\|\mathbf{x}\| > \|\mathbf{b}\|$ (where $\|\mathbf{x}\| = \sum_{i=1}^{d} \mathbf{x}(i)$), then a necessary condition for reachability of $\mathbf{b}$ from $\mathbf{x}$ is that $\mathbf{x}$ belongs to the set

$$\mathrm{DC} = \{\mathbf{x} \in \mathbb{N}^d \mid \text{ there is } \mathbf{x}' \text{ such that } \mathbf{x} \xrightarrow{A^*} \mathbf{x}' \text{ and } \|\mathbf{x}\| > \|\mathbf{x}'\|\}.$$

For $\mathbf{x} \in \mathrm{DC}$ we say that $\mathbf{x}$ *can Decrease the token-Count.* Since there is no infinite sequence $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$ in $\mathbb{N}^d$ where $\|\mathbf{x}_1\| > \|\mathbf{x}_2\| > \|\mathbf{x}_3\| > \cdots$, for $\mathrm{NDC} = \overline{\mathrm{DC}}$ (the complement of DC, i.e. $\mathbb{N}^d \smallsetminus \mathrm{DC}$) we note the following trivial fact:

▶ **Observation 15.** NDC *is a home-space for every* $\mathbf{X} \subseteq \mathbb{N}^d$.

Proposition 16 is a crucial ingredient for Proposition 17 that finishes the proof of Lemma 8 in the special case when $\mathbf{L}$ is a singleton.

▶ **Proposition 16.** *The set* DC *is upward closed and the set* min(DC) *is effectively constructible. Hence both* DC *and* NDC *are effectively constructible semilinear sets.*

Proof. If $\mathbf{x} \xrightarrow{\sigma} \mathbf{x}'$, then $\mathbf{x} + \mathbf{y} \xrightarrow{\sigma} \mathbf{x}' + \mathbf{y}$ (by the monotonicity property of Petri nets). Since $\|\mathbf{x}\| > \|\mathbf{x}'\|$ entails $\|\mathbf{x} + \mathbf{y}\| > \|\mathbf{x}' + \mathbf{y}\|$, it is clear that DC is upward closed (i.e., if $\mathbf{x} \in \mathrm{DC}$ and $\mathbf{x} \leq \mathbf{y}$, then $\mathbf{y} \in \mathrm{DC}$).

Regarding the effective constructability of min(DC), we recall Lemma 13. The question if $(\mathrm{DC} \cap \downarrow\mathbf{v})$ is nonempty, for a given $\mathbf{v} \in \mathbb{N}_\omega^d$, can be reduced to the reachability problem in a standard way (recall the technique sketched for Proposition 10): in the respective net (of a bigger dimension), first some $\mathbf{y} \in \mathbb{N}^d$ belonging to $\downarrow\mathbf{v}$ is generated, and frozen, and then some $\mathbf{y}'$ reachable from $\mathbf{y}$ in the original net is obtained and frozen, and in the final phase a particular place can reach zero if, and only if, $\|\mathbf{y}\| > \|\mathbf{y}'\|$. Hence in the positive case a witness of the respective reachability also yields some $\mathbf{y} \in (\mathrm{DC} \cap \downarrow\mathbf{v})$.

The effective semilinearity of DC and NDC follows trivially. ◀

▶ **Proposition 17.** *Given a Petri net A of dimension d and a vector $\mathbf{b} \in \mathbb{N}^d$, the set*

$$\mathbf{C} = \mathrm{NDC} \cap \big(\{\mathbf{x} \in \mathbb{N}^d \mid \|\mathbf{x}\| > \|\mathbf{b}\|\} \cup \{\mathbf{x} \in \mathbb{N}^d \mid \|\mathbf{x}\| \leq \|\mathbf{b}\| \textit{ and } \mathbf{x} \overset{A^*}{\not\longrightarrow} \mathbf{b}\}\big)$$

is an effectively constructible semilinear non-reachability core for $\{\mathbf{b}\}$.

Proof. We first show that $\mathbf{C}$ is a non-reachability core for $\{\mathbf{b}\}$:

1. We have $\mathbf{C} \overset{A^*}{\not\longrightarrow} \{\mathbf{b}\}$, since $\mathbf{b}$ is clearly not reachable from any element of $\mathbf{C}$.
2. For each $\mathbf{x} \in \mathbb{N}^d$, if $\mathbf{x} \overset{A^*}{\not\longrightarrow} \mathbf{b}$, then $\mathbf{x} \xrightarrow{A^*} \mathbf{x}' \overset{A^*}{\not\longrightarrow} \mathbf{b}$ for some $\mathbf{x}' \in \mathrm{NDC}$ (recall Observation 15); the facts $\mathbf{x}' \in \mathrm{NDC}$ and $\mathbf{x}' \overset{A^*}{\not\longrightarrow} \mathbf{b}$ obviously entail $\mathbf{x}' \in \mathbf{C}$, and thus $\mathbf{x} \xrightarrow{A^*} \mathbf{C}$.

The effective semilinearity of $\mathbf{C}$ follows from Proposition 16 and from the fact that the finite set $\{\mathbf{x} \in \mathbb{N}^d \mid \|\mathbf{x}\| \leq \|\mathbf{b}\| \text{ and } \mathbf{x} \overset{A^*}{\not\longrightarrow} \mathbf{b}\}$ can be constructed by repeatedly using an algorithm deciding reachability. ◀

Now we proceed to prove Lemma 8 in general. We have a Petri net A of dimension d, and a linear set $\mathbf{L}$ presented by a basis $\mathbf{b} \in \mathbb{N}^d$ and periods $\mathbf{p}_1, \mathbf{p}_2 \ldots, \mathbf{p}_k \in \mathbb{N}^d$; we aim to construct a semilinear non-reachability core for $\mathbf{L}$. We would like to generalize the above special-case proof with the upward closed set DC, which is, in fact, closely related to the approach in [5]. But here is a subtle problem, as we already mentioned. Our solution is not working with configurations $\mathbf{x} \in \mathbb{N}^d$ directly, but rather via their $\mathbf{L}$-like presentations.

We note that each configuration $\mathbf{x} \in \mathbb{N}^d$ can be presented as

$$\mathbf{x} = \mathbf{y} + \mathbf{u}(1) \cdot \mathbf{p}_1 + \mathbf{u}(2) \cdot \mathbf{p}_2 \cdots + \mathbf{u}(k) \cdot \mathbf{p}_k$$

for at least one (but often more) pairs $(\mathbf{y}, \mathbf{u}) \in \mathbb{N}^d \times \mathbb{N}^k$. For $\mathbf{y} \in \mathbb{N}^d$ and $\mathbf{u} \in \mathbb{N}^k$ we put

$$\textsc{conf}(\mathbf{y}, \mathbf{u}) = \mathbf{y} + \mathbf{u}(1) \cdot \mathbf{p}_1 + \mathbf{u}(2) \cdot \mathbf{p}_2 \cdots + \mathbf{u}(k) \cdot \mathbf{p}_k.$$

Hence $\mathbf{L} = \{\textsc{conf}(\mathbf{b}, \mathbf{u}) \mid \mathbf{u} \in \mathbb{N}^k\}$.

Let $\textsc{dcb-pr}$ (determined by the Petri net A and the sequence of periods of $\mathbf{L}$) be the set of presentation pairs that present configurations that *can Decrease the token-Count in the presentation Basis*:

$$\textsc{dcb-pr} = \{(\mathbf{y}, \mathbf{u}) \in \mathbb{N}^d \times \mathbb{N}^k \mid \exists (\mathbf{y}', \mathbf{u}') : \|\mathbf{y}\| > \|\mathbf{y}'\|, \textsc{conf}(\mathbf{y}, \mathbf{u}) \xrightarrow{A^*} \textsc{conf}(\mathbf{y}', \mathbf{u}')\}.$$

We note that if $\mathbf{y} \geq \mathbf{p}_i$, for some $i \in [1, k]$, then we trivially have $(\mathbf{y}, \mathbf{u}) \in \textsc{dcb-pr}$ since $\textsc{conf}(\mathbf{y}, \mathbf{u}) = \textsc{conf}(\mathbf{y} - \mathbf{p}_i, \mathbf{u}')$ where $\mathbf{u}'$ arises from $\mathbf{u}$ by adding 1 to $\mathbf{u}(i)$. (As expected, we assume that all $\mathbf{p}_i$ are nonzero vectors.)

▶ **Proposition 18.** $\textsc{dcb-pr}$ *is upward closed and the set* $\min(\textsc{dcb-pr})$ *is effectively constructible.*

Proof. As expected, we compare the elements of $\textsc{dcb-pr}$ component-wise. To show that $\textsc{dcb-pr}$ is upward closed, we assume that $(\mathbf{y}_1, \mathbf{u}_1) \in \textsc{dcb-pr}$ and $(\mathbf{y}_1, \mathbf{u}_1) \leq (\mathbf{y}_2, \mathbf{u}_2)$. To demonstrate that $(\mathbf{y}_2, \mathbf{u}_2) \in \textsc{dcb-pr}$ as well, we again use monotonicity of Petri nets: Since $\textsc{conf}(\mathbf{y}_1, \mathbf{u}_1) \xrightarrow{\sigma} \textsc{conf}(\mathbf{y}'_1, \mathbf{u}'_1)$ (for some sequence σ) where $\|\mathbf{y}_1\| > \|\mathbf{y}'_1\|$, and $\textsc{conf}(\mathbf{y}_1, \mathbf{u}_1) \leq \textsc{conf}(\mathbf{y}_2, \mathbf{u}_2)$, we have $\textsc{conf}(\mathbf{y}_2, \mathbf{u}_2) \xrightarrow{\sigma} \textsc{conf}(\mathbf{y}'_1 + (\mathbf{y}_2 - \mathbf{y}_1), \mathbf{u}'_1 + (\mathbf{u}_2 - \mathbf{u}_1))$; $\|\mathbf{y}_1\| > \|\mathbf{y}'_1\|$ entails $\|\mathbf{y}_2\| > \|\mathbf{y}'_1 + (\mathbf{y}_2 - \mathbf{y}_1)\|$.

The effective constructability of $\min(\textsc{dcb-pr})$ is again based on Lemma 13, when we identify $\mathbb{N}^d \times \mathbb{N}^k$ with $\mathbb{N}^{d+k}$. It is again a technical routine to show that the question whether $(\textsc{dcb-pr} \cap \downarrow \mathbf{v})$ is nonempty, for a given $\mathbf{v} \in \mathbb{N}_\omega^{d+k}$, can be reduced to the reachability problem, so that in the positive case a witness of this reachability also yields some $(\mathbf{y}, \mathbf{u}) \in (\textsc{dcb-pr} \cap \downarrow \mathbf{v})$. ◀

We now define the set of configurations that can be presented so that the presentation basis cannot be decreased:

$$\text{NDCB} = \{\mathbf{x} \in \mathbb{N}^d \mid \mathbf{x} = \textsc{conf}(\mathbf{y}, \mathbf{u}) \text{ for some } (\mathbf{y}, \mathbf{u}) \notin \textsc{dcb-pr}\}.$$

▶ **Observation 19.** NDCB *is a home-space for every* $\mathbf{X} \subseteq \mathbb{N}^d$.

(Suppose there is some $\mathbf{x} \in \mathbb{N}^d$ such that $\mathbf{x} \overset{A^*}{\not\longrightarrow}$ NDCB; we fix one such $\mathbf{x}$ that can be written as $\mathbf{x} = \textsc{conf}(\mathbf{y}, \mathbf{u})$ for $\mathbf{y}$ with the least norm $\|\mathbf{y}\|$. Since $\mathbf{x} \notin$ NDCB, we have $(\mathbf{y}, \mathbf{u}) \in \textsc{dcb-pr}$, which entails a contradiction by the definition of $\textsc{dcb-pr}$.)

▶ **Proposition 20.** NDCB *is an effectively constructible semilinear set.*

Proof. By Proposition 18, $\textsc{dcb-pr}$ is an effectively constructible semilinear set. Since semilinear sets (effectively) coincide with the sets definable in Presburger arithmetic, the claim is clear. ◀

The next proposition finishes a proof of Lemma 8, and thus also of Theorem 2.

▶ **Proposition 21.** *Given a Petri net A of dimension d and a linear set $\mathbf{L} \subseteq \mathbb{N}^d$ presented by $(\mathbf{b}, \mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_k)$, the set*

$$\mathbf{C} = \{\mathbf{x} \in \mathbb{N}^d \mid \mathbf{x} = \textsc{conf}(\mathbf{y}, \mathbf{u}) \textit{ where } (\mathbf{y}, \mathbf{u}) \notin \textsc{dcb-pr} \textit{ and}$$
$$\textit{either } \|\mathbf{y}\| > \|\mathbf{b}\|, \textit{ or } \|\mathbf{y}\| \leq \|\mathbf{b}\| \textit{ and } \textsc{conf}(\mathbf{y}, \mathbf{u}) \overset{A^*}{\not\longrightarrow} \mathbf{L}\}.$$

is an effectively constructible semilinear non-reachability core for $\mathbf{L}$.

Proof. We note that $\mathbf{C}$ is a subset of NDCB, and we recall that $\mathbf{x} \in \mathbf{L}$ iff $\mathbf{x} = \textsc{conf}(\mathbf{b}, \mathbf{u})$ for some $\mathbf{u} \in \mathbb{N}^k$. We verify that $\mathbf{C}$ is a non-reachability core for $\mathbf{L}$:

1. By definition of $\mathbf{C}$ we clearly have $\mathbf{C} \overset{A^*}{\not\longrightarrow} \mathbf{L}$.
2. For each $\mathbf{x} \in \mathbb{N}^d$, if $\mathbf{x} \overset{A^*}{\not\longrightarrow} \mathbf{L}$, then $\mathbf{x} \xrightarrow{A^*} \mathbf{x}' \overset{A^*}{\not\longrightarrow} \mathbf{L}$ for some $\mathbf{x}' \in$ NDCB (recall Observation 19); the facts $\mathbf{x}' \in$ NDCB and $\mathbf{x}' \overset{A^*}{\not\longrightarrow} \mathbf{L}$ obviously entail $\mathbf{x}' \in \mathbf{C}$, and thus $\mathbf{x} \xrightarrow{A^*} \mathbf{C}$.

Now we aim to show that $\mathbf{C}$ is an effectively constructible semilinear set. We recall Propositions 20 and 18, and the fact that for any concrete $\mathbf{y}$ and $\mathbf{u}$ we can decide if $\textsc{conf}(\mathbf{y}, \mathbf{u}) \xrightarrow{A^*} \mathbf{L}$. Though there are only finitely many $\mathbf{y}$ to consider, namely those satisfying $\|\mathbf{y}\| \leq \|\mathbf{b}\|$, we are not done: it is not immediately obvious how to express $\textsc{conf}(\mathbf{y}, \mathbf{u}) \overset{A^*}{\not\longrightarrow} \mathbf{L}$ in Presburger arithmetic, even when $\mathbf{y}$ is fixed. To this aim, for any fixed $\mathbf{y} \in \mathbb{N}^d$ we define the set

$$\mathbf{U}_\mathbf{y} = \{\mathbf{u} \in \mathbb{N}^k \mid \textsc{conf}(\mathbf{y}, \mathbf{u}) \xrightarrow{A^*} \mathbf{L}\} = \{\mathbf{u} \in \mathbb{N}^k \mid \exists \mathbf{u}' \in \mathbb{N}^k : \textsc{conf}(\mathbf{y}, \mathbf{u}) \xrightarrow{A^*} \textsc{conf}(\mathbf{b}, \mathbf{u}')\}.$$

For each fixed $\mathbf{y} \in \mathbb{N}^d$, the set $\mathbf{U}_\mathbf{y}$ is clearly upward closed (by monotonicity of Petri nets). Moreover, the set $\min(\mathbf{U}_\mathbf{y})$ is effectively constructible, again by using Lemma 13: Given a fixed $\mathbf{y}$, for each $\mathbf{v} \in \mathbb{N}^k_\omega$ we can decide whether $(\mathbf{U}_\mathbf{y} \cap {\downarrow}\mathbf{v})$ is nonempty by a reduction to the reachability problem, so that in the positive case a witness of this reachability also yields some $\mathbf{u} \in (\mathbf{U}_\mathbf{y} \cap {\downarrow}\mathbf{v})$.

Now it is clear that we can effectively construct a Presburger formula defining $\mathbf{C}$; hence $\mathbf{C}$ is a semilinear set for which we can effectively construct a presentation. ◀

6 Minimal Reachable Configurations

In this section we provide several Ackermannian-time algorithms. The first one is given a Petri net A of dimension d and a configuration $\mathbf{x} \in \mathbb{N}^d$, and it computes the set $\min(\textsc{post}^*_A(\mathbf{x}))$, i.e. the set of minimal configurations in the respective reachability set. The second algorithm computes $\min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{S})$ when given (a presentation of) a semilinear set $\mathbf{S} \subseteq \mathbb{N}^d$ besides A and $\mathbf{x}$. The third algorithm is given $A, \mathbf{x}$, and (a presentation of) a semilinear predicate $P \subseteq \mathbb{N}^h \times \mathbb{N}^d \times \mathbb{N}^d$ (for some $h \in \mathbb{N}$), and it computes the set

$$\min(\{\mathbf{x} \in \mathbb{N}^h \mid \exists \alpha, \beta \in \mathbb{N}^d : \alpha \xrightarrow{A^*} \beta \wedge (\mathbf{x}, \alpha, \beta) \in P\}).$$

The complexity of computing the above mentioned minimal configurations can be derived by using the approach by Hsu-Chun Yen and Chien-Liang Chen in [21]; they observed that complexity bounds on a set-related algorithm related to some set $\mathbf{X} \subseteq \mathbb{N}^d$ (recall the definition before Lemma 13) allow us to derive complexity bounds on the computation of $\min(\mathbf{X})$. As a crucial ingredient here, we recall the known complexity upper bound for reachability in Section 6.1. In Section 6.2 we derive an Ackermannian bound on the size of minimal configurations in Petri net reachability sets, and we extend this bound in Section 6.3 and in Section 6.4 to obtain the mentioned second algorithm and the third algorithm, respectively.

▶ Remark 22. Mayr and Meyer described in [17] a family of Petri nets that exhibits finite reachability sets whose size grows as the Ackermann function; hence also the size of the maximal configurations in these sets grows similarly. Concerning the size of minimal configurations, we cannot deduce any interesting size properties using the same family. However, by using the family of Petri nets recently introduced in [14, 4, 12] for proving that the reachability problem is Ackermann-hard, we can observe that the maximal size of minimal configurations in Petri net reachability sets grows at least as the Ackermann function.

6.1 Petri Net Reachability Problem in Fixed Dimension

Here we recall some definitions in order to state that the Petri net reachability problem is primitive-recursive when restricted to a fixed dimension, and Ackermannian in general.

The *fast-growing functions* $F_d : \mathbb{N} \to \mathbb{N}$, $d \in \mathbb{N}$, are defined inductively: $F_0(n) = n + 1$, and $F_{d+1}(n) = F_d^{(n+1)}(n)$; by $f^{(n)}$, for a function $f : \mathbb{N} \to \mathbb{N}$, we denote the iteration of f by itself n times (i.e., $f^{(n+1)} = f^{(n)} \circ f$). Following [18], we introduce the class $\mathbf{F}_d$ of functions computable in time $O(F_d(F_{d-1}^{(c)}(n)))$ where n is the size of the input and $c \in \mathbb{N}$ is any constant. We recall that $\bigcup_{d\in\mathbb{N}} \mathbf{F}_d$ is the class of *primitive-recursive functions*. We also introduce the function $F_\omega : \mathbb{N} \to \mathbb{N}$ defined by $F_\omega(n) = F_n(n)$, which is a variant of the Ackermann function; by $\mathbf{F}_\omega$ we denote the class of functions computable in time $O(F_\omega(F_d(n)))$ where $d \in \mathbb{N}$ is any constant and n is the size of the input. A function in $\mathbf{F}_\omega$ is said to be computable in *Ackermannian time*. (We note that Ackermannian time coincides with Ackermannian space.)

For $\mathbf{x} \in \mathbb{N}^d$ we have defined the norm of $\mathbf{x}$ as $\|\mathbf{x}\| = \sum_{i=1}^d \mathbf{x}(i)$. Now we extend the notion of norm to other objects. For a *Petri net action* $a = (\mathbf{a}_-, \mathbf{a}_+)$, by its *norm* we mean $\|a\| = \max\{\|\mathbf{a}_-\|, \|\mathbf{a}_+\|\}$. For a *Petri net* A, by its *norm* we mean $\|A\| = \max_{a\in A} \|a\|$. The *norm* of a linear set $\mathbf{L} \subseteq \mathbb{N}^d$ implicitly given by a presentation $(\mathbf{b}, \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k)$ is defined by $\|\mathbf{L}\| = \max\{\|\mathbf{b}\|, \|\mathbf{p}_1\|, \|\mathbf{p}_2\|, \ldots, \|\mathbf{p}_k\|\}$. The *norm* of a semilinear set $\mathbf{S} \subseteq \mathbb{N}^d$ implicitly given by a sequence of presentations of $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_m$ is defined by $\|\mathbf{S}\| = \max_{1\le n\le m} \|\mathbf{L}_n\|$.

Now we recall a result showing that the reachability problem restricted to Petri nets of dimension d is in $\mathbf{F}_{d+4}$, and that the general Petri net reachability problem is in $\mathbf{F}_\omega$. (We view a decision problem as a function with the co-domain $\{0, 1\}$.) This result is crucial for us to derive the upper bound in Theorem 3.

▶ **Theorem 23** ([15]). *There is a constant $c > 0$ such that for all $d, n, A, \mathbf{x}, \mathbf{y}$ where $d, n \in \mathbb{N}$, A is a Petri net of dimension d, $\mathbf{x}, \mathbf{y} \in \mathbb{N}^d$, and the norms of $A, \mathbf{x}, \mathbf{y}$ are bounded by n, we have that if $\mathbf{x} \xrightarrow{A^*} \mathbf{y}$, then $\mathbf{x} \xrightarrow{\sigma} \mathbf{y}$ for a word $\sigma \in A^*$ such that $|\sigma| \le F_{d+4} \circ F_{d+3}^{(c)}(n)$.*

We remark that in what follows we formulate some results in the form

"There is a constant $c' > 0$ such that..."

Naturally we could replace c' with c without changing the meaning of the respective statements, but we prefer keeping the difference in order to highlight the special role of the constant c introduced in Theorem 23.

6.2 Minimal Reachable Configurations

We provide an algorithm computing the set of minimal reachable configurations, by following the approach of [21]. To ease notation, we introduce the functions $f_d = F_{d+4} \circ F_{d+3}^{(c)}$ $(d \in \mathbb{N})$ where c is the constant introduced in Theorem 23, and we first prove the following proposition; for $\mathbf{v} \in \mathbb{N}_\omega^d$, by its *norm* we mean $\|\mathbf{v}\| = \sum_{i:\mathbf{v}(i)\neq\omega} \mathbf{v}(i)$.

► **Proposition 24.** *For all d, n, A, $\mathbf{x}$, $\mathbf{v}$, where $d, n \in \mathbb{N}$, A is a Petri net of dimension d, $\mathbf{x} \in \mathbb{N}^d$, $\mathbf{v} \in \mathbb{N}_\omega^d$, and the norms of $A, \mathbf{x}, \mathbf{v}$ are bounded by n, we have that if $(\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{v})$ is nonempty, then there is $\mathbf{y} \in (\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{v})$ such that $\mathbf{x} \xrightarrow{\sigma} \mathbf{y}$ for some $\sigma \in A^*$ where $|\sigma| \leq f_d(n)$.*

Proof. For $n = 0$ the claim is trivial, so we assume $n \geq 1$.

For each $j \in [1, d]$ we define the Petri net action $b_j = (\mathbf{i}_j, \mathbf{0})$ where $\mathbf{i}_j(j) = 1$ and $\mathbf{i}_j(i) = 0$ for all $i \in [1, d] \setminus \{j\}$; this action decrements the jth component of configurations. We put $I_\omega = \{j \mid j \in [1, d], \mathbf{v}(j) = \omega\}$, and by B we denote the Petri net $\{b_j \mid j \in I_\omega\}$. Since $n \geq 1$, we derive $\|A \cup B\| \leq n$.

Let us now assume a configuration $\mathbf{z} \in (\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{v})$. Let $\mathbf{c}$ be the configuration arising from $\mathbf{z}$ by replacing the components in I_ω with zero; we thus have $\|\mathbf{c}\| \leq \|\mathbf{v}\| \leq n$ (using the fact that $\mathbf{c} \leq \mathbf{z}$, and thus $\mathbf{c} \in \downarrow\mathbf{v}$).

From $\mathbf{x} \xrightarrow{A^*} \mathbf{z}$ and $\mathbf{z} \xrightarrow{B^*} \mathbf{c}$ we derive $\mathbf{x} \xrightarrow{(A\cup B)^*} \mathbf{c}$. By Theorem 23 we deduce that $\mathbf{x} \xrightarrow{u} \mathbf{c}$ for some word $u \in (A \cup B)^*$ for which $|u| \leq f_d(n)$. Since Petri net actions in B only decrease some components, we can assume that all these actions in u are at the end; hence $u = \sigma v$ where $\sigma \in A^*$ and $v \in B^*$, and we have $\mathbf{x} \xrightarrow{\sigma} \mathbf{y} \xrightarrow{v} \mathbf{c}$ for a configuration $\mathbf{y} \in \textsc{post}_A^*(\mathbf{x})$. Since $\mathbf{c} \leq \mathbf{z}$, $\mathbf{z} \in\downarrow\mathbf{v}$, and $\mathbf{y} \xrightarrow{v} \mathbf{c}$ only decreases the components that are ω in $\mathbf{v}$, we deduce that $\mathbf{y} \in\downarrow\mathbf{v}$. ◄

To ease the formulation of the next proposition, we define the functions $g_d : \mathbb{N} \to \mathbb{N}$ by $g_d(n) = n \cdot (\,2 + f_d(n)\,)$, for all $d \in \mathbb{N}$.

► **Proposition 25.** *For all $d, n, A, \mathbf{x}, \mathbf{v}, \mathbf{m}$, where $d, n \in \mathbb{N}$, A is a Petri net of dimension d, $\mathbf{x} \in \mathbb{N}^d$, $\mathbf{v} \in \mathbb{N}_\omega^d$, $\mathbf{m}$ belongs to $\min(\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{v})$, and the norms of $A, \mathbf{x}, \mathbf{v}$ are bounded by n, there exists a word $\sigma \in A^*$ such that $\mathbf{x} \xrightarrow{\sigma} \mathbf{m}$ and $|\sigma| \leq f_d \circ g_d^{(k)}(n)$ where $k = |\{i \mid \mathbf{v}(i) = \omega\}|$.*

Proof. The strict version $<$ of the relation $\leq$ on $\mathbb{N}_\omega^d$ (defined by $\mathbf{w} < \mathbf{v}$ if $\mathbf{w} \leq \mathbf{v}$ and $\mathbf{w} \neq \mathbf{v}$) is clearly well-founded. We use this property for an inductive proof.

We aim to show the claim for a considered tuple $d, n, A, \mathbf{x}, \mathbf{v}, \mathbf{m}$, while we can assume that the claim is valid for $d, n', A, \mathbf{x}, \mathbf{w}, \mathbf{m}'$ for all $\mathbf{w} < \mathbf{v}$ and all $\mathbf{m}' \in \min(\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{w})$.

Since $\mathbf{m}$ is in $(\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{v})$, we deduce from Proposition 24 that we can fix $\mathbf{y} \in (\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{v})$ and a word $\sigma \in A^*$ such that $\mathbf{x} \xrightarrow{\sigma} \mathbf{y}$ and $|\sigma| \leq f_d(n)$; we thus have $\|\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{A}\| \cdot |\sigma| \leq g_d(n) - n$. If $\mathbf{m} = \mathbf{y}$, then the claim is proved; so we assume that $\mathbf{m} \neq \mathbf{y}$.

By Observation 12 we can fix $\mathbf{w} \in \delta_{\mathbf{y}}(\mathbf{v})$ such that $\mathbf{m} \in \min(\textsc{post}_A^*(\mathbf{x}) \cap \downarrow\mathbf{w})$; since $\mathbf{w} \in \delta_{\mathbf{y}}(\mathbf{v})$, we have $\mathbf{w} < \mathbf{v}$. By the induction hypothesis, there is a word $\sigma' \in A^*$ such that $\mathbf{x} \xrightarrow{\sigma'} \mathbf{m}$ and $|\sigma'| \leq f_d \circ g_d^{(k')}(n')$ where $n' = \max\{\|A\|, \|\mathbf{x}\|, \|\mathbf{w}\|\})$ and $k' = |\{i \mid \mathbf{w}(i) = \omega\}|$.

Putting $k = |\{i \mid \mathbf{v}(i) = \omega\}|$, we observe that $k' = k$ or $k' = k - 1$. If $k' = k$, then $\|\mathbf{w}\| < \|\mathbf{v}\|$ and we are done by monotonicity of f_d and g_d. Otherwise $k' = k - 1$ and in that case $\|\mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{y}\| \leq g_d(n)$ since in that case $\mathbf{w}$ is obtained from $\mathbf{v}$ by replacing component i of $\mathbf{v}$ for some i such that $\mathbf{v}(i) = \omega$ and $\mathbf{y}(i) > 0$ by $\mathbf{y}(i) - 1$. It follows that $n' \leq g_d(n)$ and we are done also in that case by monotonicity of f_d and g_d. ◀

Finally, by instantiating the previous proposition with $\mathbf{v} = (\omega, \omega, \ldots, \omega)$, and by bounding $f_d \circ g_d^{(d)}(n)$ by $F_{d+5}(c'n)$ for some constant $c' > 0$ independent of d, n, we deduce the following two corollaries.

▶ **Corollary 26.** *There is a constant $c' > 0$ such that for all $d, n, A, \mathbf{x}, \mathbf{m}$, where $d, n \in \mathbb{N}$, A is a Petri net of dimension d, $\mathbf{x} \in \mathbb{N}^d$, $\mathbf{m}$ belongs to $\min(\textsc{post}^*_A(\mathbf{x}))$, and the norms of $A, \mathbf{x}$ are bounded by n, there exists a word $\sigma \in A^*$ such that $\mathbf{x} \xrightarrow{\sigma} \mathbf{m}$ and $|\sigma| \leq F_{d+5}(c'n)$.*

▶ **Corollary 27.** *There is a constant $c' > 0$ such that for all $d, n, A, \mathbf{x}$, where $d, n \in \mathbb{N}$, A is a Petri net of dimension d, $\mathbf{x} \in \mathbb{N}^d$, and the norms of $A, \mathbf{x}$ are bounded by n, the set $\min(\textsc{post}^*_A(\mathbf{x}))$ is computable in time exponential in $F_{d+5}(c'n)$ and the norms of vectors in that set are bounded by $n \cdot (1 + F_{d+5}(c'n))$.*

Proof. In fact, the set of minimal reachable configurations can be obtained by exploring configurations reachable from $\mathbf{x}$ by sequences of at most $F_{d+5}(c'n)$ actions in A. We note that the norms of configurations reachable in this way are bounded by $\|\mathbf{x}\| + F_{d+5}(c'n) \cdot \|A\| \leq n \cdot (1 + F_{d+5}(c'n))$. ◀

6.3 Extension to Semilinear Sets

The algorithm computing minimal reachable configurations can be also simply used for computing the set $\min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{S})$ where $\mathbf{S}$ is a semilinear set; we thus formulate this fact as a corollary (though with providing a proof). We recall that the norm of a semilinear set is the maximum norm of vectors occurring in its (implicitly assumed) presentation.

▶ **Corollary 28.** *There is a constant $c' > 0$ such that for all $d, n, A, \mathbf{x}, \mathbf{S}$, where $d, n \in \mathbb{N}$, A is a Petri net of dimension d, $\mathbf{x} \in \mathbb{N}^d$, $\mathbf{S}$ is (a presentation of) a semilinear set $\mathbf{S} \subseteq \mathbb{N}^d$, and the norms of $A, \mathbf{x}, \mathbf{S}$ are bounded by n, the set $\min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{S})$ is computable in time exponential in $F_{2d+6}(c'n)$ and the norms of vectors in that set are bounded by $n \cdot (1 + F_{2d+6}(c'n))$.*

Proof. Let us consider a d-dimensional Petri net A, an initial configuration $\mathbf{x}$, and a semilinear set $\mathbf{S} \subseteq \mathbb{N}^d$ given as the union of linear sets $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_m$. Since $\min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{S}) = \min(\bigcup_{j=1}^{m} \min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{L}_j))$ we can reduce the problem of computing $\min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{S})$ to the special case of a linear set $\mathbf{S}$, denoted as $\mathbf{L}$ in the sequel. So, let $\mathbf{L}$ be a linear set presented by a basis $\mathbf{b} \in \mathbb{N}^d$ and a sequence of periods $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \in \mathbb{N}^d$, and let us provide an algorithm for computing $\min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{L})$.

To do so, we build from A a new Petri net B of dimension $2d+1$ defined as follows and an initial configuration $(\mathbf{x}, 1, \mathbf{0})$. We associate to each Petri net action $a \in A$ of the form $(\mathbf{a}_-, \mathbf{a}_+)$ the action $((\mathbf{a}_-, 1, \mathbf{0}), (\mathbf{a}_+, 1, \mathbf{0}))$ in B that intuitively executes a on the first d counters and check that the middle counter (the counter $d+1$) is at least 1. We also add in B for each $j \in [1, k]$ an action $((\mathbf{p}_j, 0, \mathbf{0}), (\mathbf{0}, 0, \mathbf{p}_j))$ that removes the period $\mathbf{p}_j$ on the first d counters and adds it on the last d counters. Finally, we add to B the action $((\mathbf{b}, 1, \mathbf{0}), (\mathbf{0}, 0, \mathbf{b}))$ that decrements the middle counter and simultaneously removes $\mathbf{b}$ from the first d counters, and adds $\mathbf{b}$ on the last d counters. Since for any set $\mathbf{X} \subseteq \mathbb{N}^d$ and any set $I \subseteq [1, d]$, the set $\min(\{\mathbf{x} \in \mathbf{X} \mid \bigwedge_{i \in I} \mathbf{x}(i) = 0\})$ is equal to $\{\mathbf{m} \in \min(\mathbf{X}) \mid \bigwedge_{i \in I} \mathbf{m}(i) = 0\}$), one can observe that $\{\mathbf{0}\} \times \{0\} \times \min(\textsc{post}^*_A(\mathbf{x}) \cap \mathbf{L})$ is equal to $\min(\textsc{post}^*_B(\mathbf{x}, 1, \mathbf{0})) \cap (\{\mathbf{0}\} \times \{0\} \times \mathbb{N}^d)$. ◀

6.4 Extension to Semilinear Predicates

By another corollary (with a proof) we also note that the algorithm computing minimal reachable configurations can be used for computing minimal vectors in sets of the following form

$$\mathbf{X} = \{\mathbf{x} \in \mathbb{N}^h \mid \exists \alpha, \beta \in \mathbb{N}^d : \alpha \xrightarrow{A^*} \beta \wedge (\mathbf{x}, \alpha, \beta) \in P\} \tag{3}$$

where $P \subseteq \mathbb{N}^h \times \mathbb{N}^d \times \mathbb{N}^d$ is a semilinear predicate given by a presentation. Notice that we use Greek letters α and β in the definition of $\mathbf{X}$ in order to emphasise vectors that act as configurations of the Petri net A.

▶ **Corollary 29.** *There is a constant $c' > 0$ such that for all d, h, n, A, P, where $d, h, n \in \mathbb{N}$, A is a Petri net of dimension d, $\mathbf{x} \in \mathbb{N}^d$, P is (a presentation of) a semilinear predicate $P \subseteq \mathbb{N}^h \times \mathbb{N}^d \times \mathbb{N}^d$, and the norms of $A, \mathbf{x}, P$ are bounded by n, the set of minimal elements of the set $\mathbf{X}$ denoted by equation (3) is computable in time exponential in $F_{2h+4d+6}(c'n)$ and the norms of these minimal elements are bounded by $n \cdot (1 + F_{2h+4d+6}(c'n))$.*

Proof. We first introduce the set Y defined as $Z \cap P$ where

$$Z = \{(\mathbf{x}, \alpha, \beta) \in \mathbb{N}^h \times \mathbb{N}^d \times \mathbb{N}^d \mid \alpha \xrightarrow{A^*} \beta\}.$$

Since $\min(\mathbf{X}) = \min\{\mathbf{x} \in \mathbb{N}^k \mid \exists \alpha, \beta \in \mathbb{N}^d : \ (\mathbf{x}, \alpha, \beta) \in \min(Y)\}$ it is sufficient to provide an algorithm computing $\min(Y)$.

Our algorithm is based on the fact that Z is the reachability set of a $(h + 2d)$-dimensional Petri net B starting from the zero configuration and defined as follows from A. By $\mathbf{i}_i$ we denote the vector in $\mathbb{N}^h$ defined by $\mathbf{i}_i(i) = 1$ and $\mathbf{i}_i(j) = 0$ if $j \in [1, h] \backslash \{i\}$. The Petri net B is defined as the actions $((\mathbf{0}, \mathbf{0}, \mathbf{0}), (\mathbf{i}_j, \mathbf{0}, \mathbf{0}))$ where $j \in [1, h]$ that increment the counters corresponding to $\mathbf{x}$, actions $((\mathbf{0}, \mathbf{0}, \mathbf{0}), (\mathbf{0}, \mathbf{i}_j, \mathbf{i}_j))$ that increment simultaneously by the same amount the counters corresponding to α and β, and actions obtained from A that simulate the computation of A on the counters β and defined for each action a of A of the form $(\mathbf{a}_-, \mathbf{a}_+)$ by the action $((\mathbf{0}, \mathbf{0}, \mathbf{a}_-), (\mathbf{0}, \mathbf{0}, \mathbf{a}_+))$ in B. Notice that $Z = \textsc{post}^*_B(\mathbf{0}, \mathbf{0}, \mathbf{0})$ and we are done by Corollary 28. ◀

7 Complexity of the Semilinear Home-Space Problem

In this section we provide an Ackermannian complexity upper-bound for deciding the semilinear home-space problem; Theorem 3 will thus be proven.

So let $A, \mathbf{X}, \mathbf{H}$ be an instance of the semilinear home-space problem where A is a Petri net, of dimension d, and $\mathbf{X}, \mathbf{H}$ are two (presentations of) semilinear subsets of $\mathbb{N}^d$. Since $\mathbf{H}$ can be decomposed, in elementary time, into a finite union of linear sets using presentations with at most d periods [7, Lemma 6.6], we can assume that each linear set $\mathbf{L}$ of the presentation of $\mathbf{H}$ satisfies this constraint. We put $n = \max\{\|A\|, \|\mathbf{X}\|, \|\mathbf{H}\|\}$.

We first consider the problem of computing a semilinear non-reachability core for each linear set $\mathbf{L}$ of the presentation of $\mathbf{H}$. Such a linear set $\mathbf{L}$ is presented with a basis $\mathbf{b}$ and a sequence of k periods $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k$ with $k \leq d$. As previously shown, this computation reduces to the computation of the minimal elements of the upward closed set DCB-PR and the upward-closed sets $\mathbf{U}_\mathbf{y}$ where $\mathbf{y}$ belongs to the finite set of vectors in $\mathbb{N}^d$ satisfying $\|\mathbf{y}\| \leq \|\mathbf{b}\|$. The computation of those minimal elements can be obtained by rewriting the definitions of DCB-PR and $\mathbf{U}_\mathbf{y}$ to match the statement of Corollary 29. To do so, we note that DCB-PR and $\mathbf{U}_\mathbf{y}$ can be described in the following way:

$$\textsc{dcb-pr} = \{(\mathbf{y},\mathbf{u}) \in \mathbb{N}^d \times \mathbb{N}^k \mid \exists \alpha, \beta \in \mathbb{N}^d : \alpha \xrightarrow{A^*} \beta \wedge (\mathbf{y},\mathbf{u},\alpha,\beta) \in P\}$$
$$\mathbf{U}_{\mathbf{y}} = \{\mathbf{u} \in \mathbb{N}^k \mid \exists \alpha, \beta \in \mathbb{N}^d : \alpha \xrightarrow{A^*} \beta \wedge (\mathbf{u},\alpha,\beta) \in P_{\mathbf{y}}\}$$

where:

$$P = \left\{ (\mathbf{y},\mathbf{u},\alpha,\beta) \in \mathbb{N}^d \times \mathbb{N}^k \times \mathbb{N}^d \times \mathbb{N}^d \mid \exists (\mathbf{y}',\mathbf{u}') \in \mathbb{N}^d \times \mathbb{N}^k : \begin{array}{l} \|\mathbf{y}\| > \|\mathbf{y}'\| \wedge \\ \alpha = \textsc{conf}(\mathbf{y},\mathbf{u}) \wedge \\ \beta = \textsc{conf}(\mathbf{y}',\mathbf{u}') \end{array} \right\}$$
$$P_{\mathbf{y}} = \{(\mathbf{u},\alpha,\beta) \in \mathbb{N}^k \times \mathbb{N}^d \times \mathbb{N}^d \mid \alpha = \textsc{conf}(\mathbf{y},\mathbf{u}) \wedge \beta \in \mathbf{L}\}.$$

Since the sets P and $P_{\mathbf{y}}$ are clearly expressible by formulas in Presburger arithmetic, we can effectively construct, in elementary time, semilinear presentations of those sets [8]. We introduce an elementary function E (independent of any instance) corresponding to that computation. We deduce that for some constant $c' > 0$, independent of any input, we can compute, in time exponential in $F_{8d+6}(c'E(n))$, the sets $\min(\textsc{dcb-pr})$ and $\min(\mathbf{U}_{\mathbf{y}})$ for $\|\mathbf{y}\| \leq \|\mathbf{b}\|$. Moreover, the norms of vectors in those sets are bounded by $F_{8d+6}(c'E(n))$. It follows from the proof of Proposition 21 that there exists an elementary function E' (independent of any instance) such that we can compute, in time $E'(F_{8d+6}(c'E(n)))$, a (presentation of a) semilinear non-reachability core $\mathbf{C}$ for each linear set $\mathbf{L}$ of the presentation of $\mathbf{H}$.

Let $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_m$ be the presentation sequence of $\mathbf{H}$, and let $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_m$ be the respective semilinear non-reachability cores computed for $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_m$, respectively, as shown in the previous paragraph. Proposition 9 shows that $\mathbf{H}$ is not a home-space for $\mathbf{X}$ if, and only if, there is an execution

$$\mathbf{x}_0 \xrightarrow{A^*} \mathbf{x}_1 \xrightarrow{A^*} \mathbf{x}_2 \cdots \xrightarrow{A^*} \mathbf{x}_m \tag{4}$$

where $\mathbf{x}_0 \in \mathbf{X}$, and $\mathbf{x}_i \in \mathbf{C}_i$ for each $i \in [1, m]$.

The existence of such an execution can be decided by Proposition 10, by a reduction to the reachability problem for a Petri net of a dimension that is elementary in $\max\{d, m, n\}$. Theorem 23 thus entails that the semilinear home-space problem is decidable in Ackermannian time, which finishes the proof of Theorem 3.

8 Concluding Remarks

There are various issues that can be elaborated and added to the presented material. One such issue was mentioned in Remark 6, dealing with strengthening the lower bound.

We can also look for positive witnesses of the home-space property; e.g., we anticipate that given a Petri net A of dimension d, and two semilinear sets $S_0, S_1 \subseteq \mathbb{N}^d$, we have $\textsc{post}^*_A(S_0) \subseteq \textsc{pre}^*_A(S_1)$ (i.e., S_1 is a home-space for S_0) iff there is a semilinear set S' such that $\textsc{post}^*_A(S_0) \subseteq S' \subseteq \textsc{pre}^*_A(S_1)$.

Best and Esparza [2] consider the "existential" home-space problem that asks, given a Petri net A of dimension d and an initial configuration $\mathbf{x}$, if there exists a singleton home-space for $\{\mathbf{x}\}$; the main result of [2] shows that this existential problem is decidable. We can consider a related problem that asks, given A and $\mathbf{x}$, if there is a semilinear home-space included in $\textsc{post}^*_A(\mathbf{x})$; currently we have no answer to the respective decidability question.

References

1 Henry G. Baker, Jr. Rabin's proof of the undecidability of the reachability set inclusion problem of vector addition systems. Technical Report Computation Structures Group Memo 79, Massachusetts Institute of Technology, Project MAC, July 1973.

2 Eike Best and Javier Esparza. Existence of home states in Petri nets is decidable. *Inf. Process. Lett.*, 116(6):423–427, 2016. doi:10.1016/j.ipl.2016.01.011.

3 Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for Petri nets is not elementary. *J. ACM*, 68(1):7:1–7:28, 2021. doi:10.1145/3422822.

4 Wojciech Czerwinski and Lukasz Orlikowski. Reachability in vector addition systems is Ackermann-complete. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1229–1240. IEEE, 2021. doi:10.1109/FOCS52979.2021.00120.

5 David de Frutos Escrig and Colette Johnen. Decidability of home space property. Technical Report LRI-503, Univ. de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique, July 1989.

6 Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets – A survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

7 Seymour Ginsburg and Edwin H. Spanier. Bounded Algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964. doi:10.2307/1994067.

8 Seymour Ginsburg and Edwin H. Spanier. Semigroups, Presburger formulas and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966. doi:10.2140/pjm.1966.16.285.

9 Michel Hack. *Decidability questions for Petri nets.* PhD thesis, MIT, 1975. URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-161.pdf.

10 Michel Hack. The equality problem for vector addition systems is undecidable. *Theor. Comput. Sci.*, 2(1):77–95, 1976. doi:10.1016/0304-3975(76)90008-6.

11 Petr Jančar. Undecidability of bisimilarity for Petri nets and some related problems. *Theor. Comput. Sci.*, 148(2):281–301, 1995. doi:10.1016/0304-3975(95)00037-W.

12 Slawomir Lasota. Improved Ackermannian lower bound for the Petri nets reachability problem. In Petra Berenbrink and Benjamin Monmege, editors, *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15-18, 2022, Marseille, France (Virtual Conference)*, volume 219 of *LIPIcs*, pages 46:1–46:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.STACS.2022.46.

13 Jérôme Leroux. Vector addition systems reachability problem (A simpler solution). In Andrei Voronkov, editor, *Turing-100 – The Alan Turing Centenary, Manchester, UK, June 22-25, 2012*, volume 10 of *EPiC Series in Computing*, pages 214–228. EasyChair, 2012. doi:10.29007/bnx2.

14 Jérôme Leroux. The reachability problem for Petri nets is not primitive recursive. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1241–1252. IEEE, 2021. doi:10.1109/FOCS52979.2021.00121.

15 Jérôme Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785796.

16 Ernst W. Mayr. An algorithm for the general Petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984. doi:10.1137/0213029.

17 Ernst W. Mayr and Albert R. Meyer. The complexity of the finite containment problem for Petri nets. *J. ACM*, 28(3):561–576, 1981. doi:10.1145/322261.322271.

18 Sylvain Schmitz. Complexity hierarchies beyond elementary. *TOCT*, 8(1):3:1–3:36, 2016. doi:10.1145/2858784.

19 Sylvain Schmitz. The complexity of reachability in vector addition systems. *SIGLOG News*, 3(1):4–21, 2016. URL: https://dl.acm.org/citation.cfm?id=2893585.

20 Rüdiger Valk and Matthias Jantzen. The residue of vector sets with applications to decidability problems in Petri nets. In Grzegorz Rozenberg, Hartmann J. Genrich, and Gérard Roucairol, editors, *Advances in Petri Nets 1984*, volume 188 of *Lecture Notes in Computer Science*, pages 234–258. Springer, 1984. doi:10.1007/3-540-15204-0_14.

21 Hsu-Chun Yen and Chien-Liang Chen. On minimal elements of upward-closed sets. *Theor. Comput. Sci.*, 410(24-25):2442–2452, 2009. doi:10.1016/j.tcs.2009.02.036.

Singly Exponential Translation of Alternating Weak Büchi Automata to Unambiguous Büchi Automata

Yong Li

University of Liverpool, UK

SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China

Sven Schewe

University of Liverpool, UK

Moshe Y. Vardi

Rice University, Houston, TX, USA

Abstract

We introduce a method for translating an alternating weak Büchi automaton (AWA), which corresponds to a Linear Dynamic Logic (LDL) formula, to an unambiguous Büchi automaton (UBA). Our translations generalise constructions for Linear Temporal Logic (LTL), a less expressive specification language than LDL. In classical constructions, LTL formulas are first translated to alternating *very weak* automata (AVAs) – automata that have only singleton strongly connected components (SCCs); the AVAs are then handled by efficient disambiguation procedures. However, general AWAs can have larger SCCs, which complicates disambiguation. Currently, the only available disambiguation procedure has to go through an intermediate construction of nondeterministic Büchi automata (NBAs), which would incur an exponential blow-up of its own. We introduce a translation from *general* AWAs to UBAs with a *singly* exponential blow-up, which also immediately provides a singly exponential translation from LDL to UBAs. Interestingly, the complexity of our translation is *smaller* than the best known disambiguation algorithm for NBAs (broadly $(0.53n)^n$ vs. $(0.76n)^n$), while the input of our construction can be exponentially more succinct.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Theory of computation → Verification by model checking

Keywords and phrases Büchi automata, unambiguous automata, alternation, weak, disambiguation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.37

Related Version *Full Version*: `https://arxiv.org/abs/2305.09966`

Funding This work has been supported in part by the EPSRC through grants EP/X021513/1 and EP/X017796/1, NSFC grant 62102407, NSF grants IIS-1527668, CCF-1704883, IIS-1830549, CNS-2016656, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office.

Acknowledgements We thank the anonymous reviewers for their valuable feedback.

1 Introduction

Automata over infinite words were first introduced by Büchi [8]. The automata used by Büchi (thus called *Büchi automata*) accept an infinite word if they have a run over the word that visits accepting states infinitely often. Nondeterministic Büchi automata (NBAs) are nowadays recognized as a standard tool for model checking transition systems against temporal specification languages like Linear Temporal Logic (LTL) [1, 11, 13, 26].

NBAs belong to a larger class of automata over infinite words, also known as ω-automata. Translations between different types of ω-automata play a central role in automata theory, and many of them have gained practical importance, too. For example, researchers have started to pay attention to a kind of automata called *alternating automata* [20, 22] in the 80s.

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 37; pp. 37:1–37:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Alternating automata not only have existential, but also *universal* branching. In alternating automata, the transition function no longer maps a state and a letter to a set of states, but to a positive Boolean formula over states. An alternating Büchi automaton accepts an infinite word if there is a run graph over the word, in which all traces visit accepting states infinitely often. Every NBA can be seen as a special type of alternating Büchi automaton (ABA), while the translation from ABAs to NBAs may incur an exponential blow-up in the number of states [20]. Indeed, ABAs can be exponentially more succinct than their counterpart NBAs [6]. Apart from their succinctness, another reason why alternating automata have become popular in our community is their tight connection to specification logics. There is a straight forward translation from Linear Dynamic Logic (LDL) [12, 25] to *alternating weak Büchi automata* (AWAs), both recognizing exactly the ω-regular languages. AWAs are a special type of ABAs in which every strongly connected component (SCC) contains either only accepting states or only rejecting states. (AWAs have also been applied to the complementation of Büchi automata [17].) Further, there is a one-to-one mapping [5, 7, 11] between LTL and *very weak* alternating Büchi automata (AVAs) [23] – special AWAs where every SCC has only one state.

Automata over infinite words with different branching mechanisms all have their place in building the foundation of automata-theoretic model checking. This paper adds another chapter to the success story of efficient automata transformations: we show how to efficiently translate AWAs to unambiguous Büchi automata (UBAs) [10], and thus also the logics that tractably reduce to AWAs, e.g., LDL. UBAs are a type of NBAs that have at most one accepting run for each word and have found applications in probabilistic verification [2][1].

Our approach can be viewed as a generalization of earlier work on the disambiguation of AVAs [4, 14]. The property of the very weakness has proven useful for disambiguation: to obtain an unambiguous generalized Büchi automaton (UGBA) from an AVA, it essentially suffices to use the nondeterministic power of the automaton to guess, in every step, the precise set of states from which the automaton accepts. There is only one correct guess (which provides unambiguity), and discharging the correctness of these guesses is straight forward. AVAs with n states can therefore be disambiguated to UGBAs with 2^n states and n accepting sets, and thus to UBAs with $n2^n$ states.

Unfortunately, this approach does not extend easily to the disambiguation of AWAs: while there would still be exactly one correct guess, the straight-forward way to discharging its correctness would involve a breakpoint construction [20], which is *not* unambiguous.

The technical contribution of this paper is to replace these breakpoint constructions by *total preorders*, and showing that there is a *unique* correct way to choose these orders. We provide two different reductions, one closer to the underpinning principles – and thus better for a classroom (cf. Section 3.4) – and a more efficient approach (cf. Section 4).

Given that we track total preorders, the worst-case complexity arises when all, or almost all, states are in the same component. To be more precise, if $\mathsf{tpo}(n)$ denotes the number of total preorders on sets with n states, then our construction provides UBAs of size $\mathcal{O}\big(\mathsf{tpo}(n)\big)$. As $\mathsf{tpo}(n) \approx \frac{n!}{2(\ln 2)^{n+1}}$ [3], we have that $\lim_{n\to\infty} \frac{\sqrt[n]{\mathsf{tpo}(n)}}{n} = \frac{1}{e \ln 2} \approx 0.53$, which is a better bound than the best known bound for Büchi disambiguation [16] (and complementation [24]), where the latter number is ≈ 0.76.

[1] We note that specialized model checking algorithm for Markov chains against AWAs/LDL, without constructing UBAs, has been proposed in [9] without implementations. Nonetheless, our translation can potentially be used as a third party tool that constructs UBA from an AWA/LDL formula for PRISM model checker [18] without changing the underlying model checking algorithm [2].

While it is not surprising that a direct construction of UBAs for AWAs is superior to a construction that goes through nondeterminization (and thus incurs two exponential blow-ups on the way), we did not initially expect a construction that leads to a smaller increase in the size when starting from an AWA compared to starting from an NBA, as AWAs can be exponentially more succinct than NBAs, but not vice versa (See [17] for a quadratic transformation).

As a final test for the quality of our construction, we briefly discuss how it behaves on AVAs, for which efficient disambiguation is available. We show that the complexity of our construction can be improved to $n2^n$ when the input is an AVA, leading to the same construction as the classic disambiguation construction for LTL/AVAs [4,14] (cf. Section 5). We also discuss how to adjust it so that it can produce the same transition based UGBA in this case, too. The greater generality we obtain comes therefore at no additional cost.

Related work. Disambiguation of AVAs from LTL specifications have been studied in [4, 14]. Our disambiguation algorithm can be seen as a more general form of them. The disambiguation of NBAs was considered in [15], which has a blow-up of $\mathcal{O}((3n)^n)$; the complexity has been later improved to $\mathcal{O}(n \cdot (0.76n)^n)$ in [16]. Our construction can also be used for disambiguating NBAs, by going through an intermediate construction of AWAs from NBAs; however, the intermediate procedure itself can incur a quadratic blow-up of states [14]. Nonetheless, if the input is an AWA, our construction improves the current best known approach exponentially by avoiding the alternation removal operation for AWAs [6,20].

2 Preliminaries

For a given set X, we denote by $\mathcal{B}^+(X)$ the set of *positive Boolean* formulas over X. These are the formulas obtained from elements of X by only using $\wedge$ and $\vee$, where we also allow $\mathtt{tt}$ and $\mathtt{ff}$. We use $\mathtt{tt}$ and $\mathtt{ff}$ to represent tautology and contradiction, respectively. For a set $Y \subseteq X$, we say Y satisfies a formula $\theta \in \mathcal{B}^+(X)$, denoted as $Y \models \theta$, if the Boolean formula θ is evaluated to $\mathtt{tt}$ when we assign $\mathtt{tt}$ to members of Y and $\mathtt{ff}$ to members of $X \setminus Y$. For an infinite sequence ρ, we denote by $\rho[i]$ the i-th element in ρ for some $i \geq 0$; for $i \in \mathbb{N}$, we denote by $\rho[i \cdots] = \rho[i]\rho[i+1]\cdots$ the suffix of ρ from its i-th letter.

An *alternating* Büchi automaton (ABA) $\mathcal{A}$ is a tuple $(\Sigma, Q, \iota, \delta, F)$ where Σ is a finite alphabet, Q is a finite set of states, $\iota \in Q$ is the initial state, $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is the transition function, and $F \subseteq Q$ is the set of accepting states. ABAs allow both non-deterministic and universal transitions. The disjunctions in transition formulas model the non-deterministic choices, while conjunctions model the universal choices. The existence of both nondeterministic and universal choices can make ABAs exponentially more succinct than NBAs [6]. We assume w.l.o.g. that every ABA is *complete*, in the sense that there is a next state for each $s \in Q$ and $\sigma \in \Sigma$. Every ABA can be made complete as follows. Fix a state $s \in Q$ and a letter $\sigma' \in \Sigma$. If $\delta(s, \sigma') = \mathtt{ff}$, we can add a sink rejecting state $\bot$, and set $\delta(s, \sigma') = \bot$ and $\delta(\bot, \sigma) = \bot$ for every $\sigma \in \Sigma$; If $\delta(s, \sigma') = \mathtt{tt}$, we can similarly add a sink accepting state $\top$, and set $\delta(s, \sigma') = \top$ and $\delta(\top, \sigma) = \top$ for every $\sigma \in \Sigma$. For a state $s \in Q$, we denote by $\mathcal{A}^s$ the ABA obtained from $\mathcal{A}$ by setting the initial state to s.

The *underlying graph* $\mathcal{G}_{\mathcal{A}}$ of an ABA $\mathcal{A}$ is a graph $\langle Q, E \rangle$, where the set of vertices is the set Q of states in $\mathcal{A}$ and $(q, q') \in E$ if q' appears in the formula $\delta(q, \sigma)$ for some $\sigma \in \Sigma$. We call a set $C \subseteq Q$ a *strongly connected component* (SCC) of $\mathcal{A}$ if, for every pair of states $q, q' \in C$, q and q' can reach each other in $\mathcal{G}_{\mathcal{A}}$.

A *nondeterministic Büchi automaton* (NBA) $\mathcal{A}$ is an ABA where $\mathcal{B}^+(Q)$ only contains the $\vee$ operator; we also allow *multiple* initial states for NBAs. We usually denote the transition function δ of an NBA $\mathcal{A}$ as a function $\delta : Q \times \Sigma \to 2^Q$ and the set of initial states as I. Let $w = w[0]w[1] \cdots \in \Sigma^\omega$ be an (infinite) *word* over Σ.

A *run* of the NBA $\mathcal{A}$ over w is a state sequence $\rho = q_0 q_1 \cdots \in Q^\omega$ such that $q_0 \in I$ and, for all $i \in \mathbb{N}$, we have that $q_{i+1} \in \delta(q_i, w[i])$. We denote by $\mathsf{inf}(\rho)$ the set of states that occur in ρ infinitely often. A run ρ of the NBA $\mathcal{A}$ is *accepting* if $\mathsf{inf}(\rho) \cap F \neq \emptyset$. An NBA $\mathcal{A}$ accepts a word w if there is an accepting run ρ of $\mathcal{A}$ over w. An NBA $\mathcal{A}$ is said to be *unambiguous* (abbreviated as UBA) [10] if $\mathcal{A}$ has at most *one* accepting run for every word.

Since ABA have universal branching (or conjunctions in δ), a run of an ABA is no longer an infinite sequence of states; instead, a run of an ABA $\mathcal{A}$ over w is a run directed acyclic graph (run DAG) $\mathcal{G}_w = (V, E)$ formally defined below:

- $V \subseteq Q \times \mathbb{N}$ where $\langle \iota, 0 \rangle \in V$.
- $E \subseteq \bigcup_{\ell > 0} (Q \times \{\ell\}) \times (Q \times \{\ell + 1\})$ where, for every vertex $\langle q, \ell \rangle \in V, \ell \geq 0$, we have that $\{ q' \in Q \mid (\langle q, \ell \rangle, \langle q', \ell + 1 \rangle) \in E \} \models \delta(q, w[\ell])$.

A vertex $\langle q, \ell \rangle$ is said to be *accepting* if $q \in F$. An infinite sequence $\rho = \langle q_0, 0 \rangle \langle q_1, 1 \rangle \cdots$ of vertices is called an *ω-branch* of $\mathcal{G}_w$ if $q_0 = \iota$ and for all $\ell \in \mathbb{N}$, we have $(\langle q_\ell, \ell \rangle, \langle q_{\ell+1}, \ell + 1 \rangle) \in E$. We also say the fragment $\langle q_i, i \rangle \langle q_{i+1}, i + 1 \rangle \cdots$ of ρ is an *ω-branch* from $\langle q_i, i \rangle$. We say a run DAG $\mathcal{G}_w$ is *accepting* if *all* its ω-branches visit accepting vertices infinitely often. An ω-word w is *accepting* if there is an accepting run DAG of $\mathcal{A}$ over w.

Let $\mathcal{A}$ be an ABA. We denote by $\mathcal{L}(\mathcal{A})$ the set of words accepted by $\mathcal{A}$.

It is known that both NBAs and ABAs recognise exactly the ω-regular languages. ABAs can be transformed into language-equivalent NBAs in exponential time [20]. In this work, we consider a special type of ABAs, called *alternating weak Büchi automata* (AWAs) where, for every SCC C of an AWA $\mathcal{A} = (\Sigma, Q, \iota, \delta, F)$, we have either $C \subseteq F$ or $C \cap F = \emptyset$. We note that different choices of equivalent transition formulas, e.g., $\delta(p, \sigma) = q_1$ and $\delta(p, \sigma) = q_1 \wedge (q_1 \vee q_2)$, will result in different SCCs. However, as long as the input ABA is weak[2], our proposed translation still applies.

One can transform an ABA to its equivalent AWA with only quadratic blow-up of the number of states [17]. A nice property of an AWA $\mathcal{A}$ is that we can easily define its dual AWA $\widehat{\mathcal{A}} = (\Sigma, Q, \iota, \widehat{\delta}, \widehat{F})$, which has the same statespace and the same underlying graph as $\mathcal{A}$, as follows: for a state $q \in Q$ and $a \in \Sigma$, $\widehat{\delta}(q, a)$ is defined from $\delta(q, a)$ by exchanging the occurrences of ff and tt and the occurrences of $\wedge$ and $\vee$, and $\widehat{F} = Q \setminus F$. It follows that:

▶ **Lemma 1** ([21]). *Let $\mathcal{A}$ be an AWA and $\widehat{\mathcal{A}}$ its dual AWA. For every state $q \in Q$, we have $\mathcal{L}(\mathcal{A}^q) = \Sigma^\omega \setminus \mathcal{L}(\widehat{\mathcal{A}}^q)$.*

In the remainder of the paper, we call a state of an NBA a *macrostate* and a run of an NBA a *macrorun* in order to distinguish them from those of ABA.

3 From AWAs to UBAs

In this section, we will present a construction of UBA $\mathcal{B}_u$ from an AWA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{B}_u) = \mathcal{L}(\mathcal{A})$. We will first introduce the construction of an NBA from an AWA given in [9] and show that this construction does *not* necessarily yield a UBA (Section 3.1). Nonetheless, we extract the essence of the construction and show that we can associate a *unique* sequence to each word (Section 3.2).

[2] To make ABAs as weak as possible, one solution would be computing minimal satisfying assignments to the transition formulas, which is well defined and results in minimal possible SCCs.

We then enrich this unique sequence with additional, similarly unique, information, which we subsequently abstract into the essence of a unique accepting macrorun of $\mathcal{B}_u$. Developing this into a UBA whose macrorun can be uniquely mapped to the sequence (Section 3.4) is then just a simple technical exercise.

3.1 From AWAs to NBAs

As shown in [20], we can obtain an equivalent NBA $\mathcal{N}(\mathcal{A})$ from an ABA $\mathcal{A}$ with an exponential blow-up of states, which is widely known as the *breakpoint construction*. In [9], the authors define a different construction of NBAs $\mathcal{B}$ from AWAs $\mathcal{A}$, which can be seen as a combination of the NBAs $\mathcal{N}(\mathcal{A})$ and $\mathcal{N}(\widehat{\mathcal{A}})$. Below we will first introduce the construction in [9] and extract its essence as a unique sequence of sets of states for each word.

The macrostate of $\mathcal{B}$ is encoded as a *consistent* tuple (Q_1, Q_2, Q_3, Q_4) such that $Q_2 = Q \setminus Q_1, Q_3 \subseteq Q_1 \setminus F$, and $Q_4 \subseteq Q_2 \setminus \widehat{F}$.

The formal translation is defined as follows.

▶ **Definition 2** ([9]). *Let $\mathcal{A} = (\Sigma, Q, \iota, \delta, F)$ be an AWA. We define an NBA $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, I_{\mathcal{B}}, \delta_{\mathcal{B}}, F_{\mathcal{B}})$ where*

- *$Q_{\mathcal{B}}$ is the set of consistent tuples over $2^Q \times 2^Q \times 2^Q \times 2^Q$.*
- *$I_{\mathcal{B}} = \{ (Q_1, Q_2, Q_3, Q_4) \in Q_{\mathcal{B}} \mid \iota \in Q_1 \}$*[3]*,*
- *Let (Q_1, Q_2, Q_3, Q_4) be a macrostate in $Q_{\mathcal{B}}$ and $\sigma \in \Sigma$.*
  Then $(Q'_1, Q'_2, Q'_3, Q'_4) \in \delta_{\mathcal{B}}((Q_1, Q_2, Q_3, Q_4), \sigma)$ if $Q'_1 \models \wedge_{s \in Q_1} \delta(s, \sigma)$ and $Q'_2 \models \wedge_{s \in Q_2} \widehat{\delta}(s, \sigma)$ and either
  - *$Q_3 = Q_4 = \emptyset, Q'_3 = Q'_1 \setminus F$ and $Q'_4 = Q'_2 \setminus \widehat{F}$,*
  - *$Q_3 \neq \emptyset$ or $Q_4 \neq \emptyset$, there exists $Y_3 \subseteq Q'_1$ such that $Y_3 \models \wedge_{s \in Q_3} \delta(s, \sigma)$ and $Q'_3 = Y_3 \setminus F$, and there exists $Y_4 \subseteq Q'_2$ such that $Y_4 \models \wedge_{s \in Q_4} \widehat{\delta}(s, \sigma)$ and $Q'_4 = Y_4 \setminus \widehat{F}$.*
- *$F_{\mathcal{B}} = \{ (Q_1, Q_2, Q_3, Q_4) \in Q_{\mathcal{B}} \mid Q_3 = Q_4 = \emptyset \}$.*

Intuitively, the resulting NBA performs two breakpoint constructions: one breakpoint construction macrostate (Q_1, Q_3) for $\mathcal{A}$ and the other breakpoint construction macrostate (Q_2, Q_4) for $\widehat{\mathcal{A}}$. Let $w \in \Sigma^\omega$. The tuple (Q_1, Q_3) in the construction uses Q_1 to keep track of the reachable states of $\mathcal{A}$ in a run DAG $\mathcal{G}_w$ over w and exploits the set Q_3 to check whether all ω-branches end in accepting SCCs. If all ω-branches in Q_3 have visited accepting vertices, Q_3 will fall empty, as Q_3 only contains non-accepting states. Once Q_3 becomes empty, we reset the set with $Q'_3 = Q'_1 \setminus F$ since we need to also check the ω-branches that newly appear in Q_1. If Q_3 becomes empty for infinitely many times, we know that every ω-branch in $\mathcal{G}_w$ is accepting, i.e., all ω-branches visit accepting vertices infinitely often. Hence w is accepted by $\mathcal{A}$ since there is an accepting run DAG from $\mathcal{A}^\iota$. We can similarly reason about the breakpoint construction for $\widehat{\mathcal{A}}$.

Besides that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A})$, Bustan, Rubin, and Vardi [9] have also shown the following:

▶ **Lemma 3** ([9]). *Let $\mathcal{B}$ be the NBA constructed as in Definition 2. Then*

- *Let $S \subseteq Q$, we have that*

$$\mathcal{L}(\mathcal{B}^{(S, Q \setminus S, Q_3, Q_4)}) = \bigcap_{s \in S} \mathcal{L}(\mathcal{A}^s) \cap \bigcap_{s \in Q \setminus S} \mathcal{L}(\widehat{\mathcal{A}}^s)$$

  where $Q_3 \subseteq S$ and $Q_4 \subseteq Q \setminus S$;

[3] $I_{\mathcal{B}}$ is not present in [9] and we added it for the completeness of the definition.

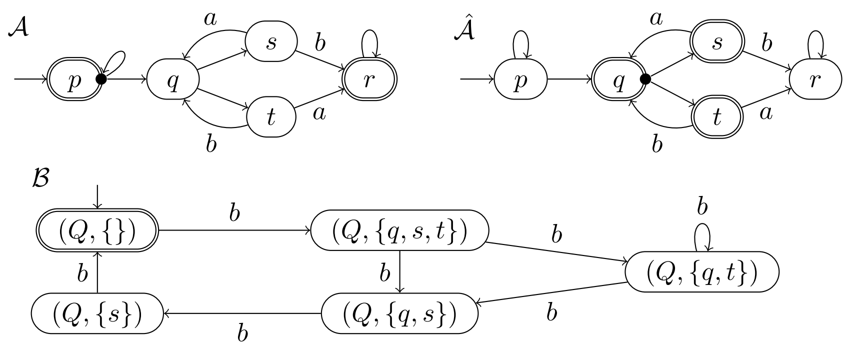


Figure 1 An example of an AWA $\mathcal{A}$, its dual $\widehat{\mathcal{A}}$ and *incomplete* part of the constructed $\mathcal{B}$ over b^ω, where for instance the transition $((Q, \{q, s\}), b, (Q, \{t\}))$ is missing.

- *Let* (Q_1, Q_2, Q_3, Q_4) *and* (Q'_1, Q'_2, Q'_3, Q'_4) *be two macrostates of* $\mathcal{B}$*, we have that*
    - $\mathcal{L}(\mathcal{B}^{(Q_1,Q_2,Q_3,Q_4)}) \cap \mathcal{L}(\mathcal{B}^{(Q'_1,Q'_2,Q'_3,Q'_4)}) = \emptyset$ *if* $Q_1 \neq Q'_1$*, and*
    - $\mathcal{L}(\mathcal{B}^{(Q_1,Q_2,Q_3,Q_4)}) = \mathcal{L}(\mathcal{B}^{(Q'_1,Q'_2,Q'_3,Q'_4)})$ *if* $Q_1 = Q'_1$*.*

Let $w \in \mathcal{L}(\mathcal{B})$ and $\rho = (Q_1^0, Q_2^0, Q_3^0, Q_4^0)(Q_1^1, Q_2^1, Q_3^1, Q_4^1) \cdots$ be an accepting macrorun of $\mathcal{B}$ over w. According to Lemma 3, it is easy to see that the Q_1-set sequence $Q_1^0 Q_1^1 \cdots$ is in fact *unique* for every accepting macrorun over w. If there are two accepting macroruns, say ρ_1 and ρ_2, of $\mathcal{B}$ over w that have two different Q_1-set sequences, there must be a position $j \geq 0$ such that their Q_1-sets differ. By Lemma 3, the suffix $w[j \cdots]$ cannot be accepted from both macrostates $\rho_1[j]$ and $\rho_2[j]$, leading to contradiction. Therefore, every accepting macrorun of $\mathcal{B}$ over w corresponds to a unique sequence of Q_1-sets. However, $\mathcal{B}$ does not necessarily have only one accepting macrorun over w, because there is *nondeterminism* in developing the breakpoints.

▶ **Lemma 4.** *The NBA* $\mathcal{B}$ *defined as in Definition 2 is not necessarily unambiguous.*

Proof. We prove Lemma 4 by giving an example AWA $\mathcal{A}$ for which the constructed $\mathcal{B}$ is *not* unambiguous. The example AWA $\mathcal{A}$ and its dual $\widehat{\mathcal{A}}$ are given in Figure 1 where accepting states are depicted with double circles, initial states are marked with an incoming arrow and universal transitions are originated from a black filled circle. The transitions are by default labelled with $\Sigma = \{a, b\}$ unless explicitly labelled otherwise. We let $Q = \{p, q, s, t, r\}$. First, we can see that $b^\omega \in \mathcal{L}(\mathcal{A}^p) \cap \mathcal{L}(\mathcal{A}^q) \cap \mathcal{L}(\mathcal{A}^s) \cap \mathcal{L}(\mathcal{A}^t) \cap \mathcal{L}(\mathcal{A}^r)$. So the unique Q_1-sequence of all accepting macroruns in $\mathcal{B}$ over b^ω should be Q^ω, according to Lemma 3. We only depict an *incomplete* part of $\mathcal{B}$ over b^ω where we ignore the Q_2 and Q_4 sets because we have constantly $Q_2 = \{\}$ and $Q_4 = \{\}$ by definition. One of the initial macrostates is $m_0 = (Q, \{\})$, which is also accepting. When reading the letter b, we always have $\{p, q, s, t, r\} \models \wedge_{c \in Q} \delta(c, b)$. Thus, the successor of m_0 over b is $m_1 = (Q, Q \setminus \{p, r\}) = (Q, \{q, s, t\})$ since the breakpoint set Q'_3 needs to be reset to $Q'_1 \setminus F$ when $Q_3 = \{\}$. When choosing the successor set Q'_3 for $Q_3 = \{q, s, t\}$ at m_1, we have two options, namely $\{q, s\}$ and $\{q, t\}$, since q has nondeterministic choices upon reading letter b. Consequently, $\mathcal{B}$ can transition to either $m_2 = (Q, \{q, s\})$ or $m_3 = (Q, \{q, t\})$, upon reading b in m_1. In fact, all the nondeterminism of $\mathcal{B}$ in Figure 1 is due to nondeterministic choices at q. We can continue to explore the state space of $\mathcal{B}$ according to Definition 2 and obtain the incomplete part of $\mathcal{B}$ depicted in Figure 1. Note that, we have ignored some outgoing transitions from $(Q, \{q, s\})$ since the present part already suffices to prove Lemma 4. It is easy to see that $\mathcal{B}$ has at least two accepting macroruns over b^ω. Thus we have proved Lemma 4. ◀

In fact, based on Definition 2, it is easy to compute a unique sequence of sets of states for each given word, which builds the foundation of our proposed construction.

3.2 Unique sequence of sets of states for each word

In the remainder of the paper, we fix an AWA $\mathcal{A} = (\Sigma, Q, \iota, \delta, F)$. For every word $w \in \Sigma^\omega$, we define a *unique* sequence of sets of states associated with it as the sequence $Q_1^0 Q_1^1 Q_1^2 \cdots$ such that, for every $i \geq 0$, we have that:

P1 $Q_1^i \subseteq Q$,
P2 for every state $q \in Q_1^i$, $w[i \cdots] \in \mathcal{L}(\mathcal{A}^q)$ and
P3 for every state $q \in Q \setminus Q_1^i$, $w[i \cdots] \notin \mathcal{L}(\mathcal{A}^q)$ (or, similarly, $w[i \cdots] \in \mathcal{L}(\widehat{\mathcal{A}}^q)$).
These properties immediately entail the weaker *local* consistency requirements:
L2 for every state $q \in Q_1^i$, $Q_1^{i+1} \models \delta(q, w[i])$ (entailed by P2) and
L3 for every state $q \in Q \setminus Q_1^i$, $Q \setminus Q_1^{i+1} \models \widehat{\delta}(q, w[i])$ (entailed by P3).

It is obvious that, for every state $s \in Q$, $\Sigma^\omega = \mathcal{L}(\mathcal{A}^s) \uplus \overline{\mathcal{L}(\mathcal{A}^s)} = \mathcal{L}(\mathcal{A}^s) \uplus \mathcal{L}(\widehat{\mathcal{A}}^s)$ holds. We define $Q_w = \{ s \in Q \mid w \in \mathcal{L}(\mathcal{A}^s) \}$. This clearly provides $Q \setminus Q_w = \{ s \in Q \mid w \in \mathcal{L}(\widehat{\mathcal{A}}^s) \}$. For every $w \in \Sigma^\omega$, we therefore have

$$w \in \bigcap_{s \in Q_w} \mathcal{L}(\mathcal{A}^s) \cap \bigcap_{s \in Q \setminus Q_w} \overline{\mathcal{L}(\mathcal{A}^s)} \text{ or, equivalently, } w \in \bigcap_{s \in Q_w} \mathcal{L}(\mathcal{A}^s) \cap \bigcap_{s \in Q \setminus Q_w} \mathcal{L}(\widehat{\mathcal{A}}^s).$$

For every $i \geq 0$, P2 and P3 are then equivalent to the requirement $Q_1^i = Q_{w[i \cdots]}$.

To see how the local constraints L2 and L3 can be obtained from P2 and P3, respectively, we fix an integer $i \geq 0$. Let $s \in Q_1^i$, so we know that $\mathcal{A}^s$ accepts $w[i \cdots]$. Let S^{i+1} be the set of successors of s in an accepting run DAG of $\mathcal{A}^s$ over $w[i \cdots]$, i.e., $S^{i+1} \models \delta(s, w[i])$. As the run DAG is accepting, we in particular have, for every $t \in S^{i+1}$, that $\mathcal{A}^t$ accepts $w[i+1 \cdots]$, which implies $S^{i+1} \subseteq Q_1^{i+1}$. With $S^{i+1} \models \delta(s, w[i])$, this provides $Q_1^{i+1} \models \delta(s, w[i])$, and thus L2.

Similarly, we can also show that, for every state $q \in Q \setminus Q_1^i$, we have $Q \setminus Q_1^{i+1} \models \widehat{\delta}(q, w[i])$. As before, $\widehat{\mathcal{A}}^q$ accepts $w[i \cdots]$ for every $q \in Q \setminus Q_1^i$ by definition. We let S^{i+1} be the set of successors of q in an accepting run DAG of $\widehat{\mathcal{A}}^q$. This implies at the same time $S^{i+1} \models \widehat{\delta}(q, w[i])$ (local constraints for the run DAG) and $S^{i+1} \subseteq Q \setminus Q_1^{i+1}$ (as the subgraphs starting there must be accepting). Together, this provides $Q \setminus Q_1^{i+1} \models \widehat{\delta}(q, w[i])$, and thus L3 also holds.

Moreover, every set Q_1^i is uniquely defined based on the word $w[i \cdots]$. Therefore, the sequence $\mathtt{R}_w = Q_1^0 Q_1^1 \cdots$ we have defined above indeed is the unique sequence satisfying P1, P2, and P3. Let us consider again the NBA construction of Definition 2: obviously, it enforces the local consistency requirements L2 and L3 on the definition of the transition relation $\delta_{\mathcal{B}}$, which is the necessary condition for the Q_1-sequence being unique; the sufficient condition that $Q_1^i = Q_{w[i \cdots]}$ for all $i \in \mathbb{N}$ is guaranteed with the two breakpoint constructions.

In the remainder of the paper, we denote this unique sequence for a given word w by $\mathtt{R}_w$. The UBA we will construct has to guess (not only) this unique sequence correctly on the fly, but also when it leaves each SCC, as shown later.

3.3 Unique distance functions

As discussed before, we have a unique sequence $\mathtt{R}_w = Q_1^0 Q_1^1 \cdots$ for w. However, as we have seen in Section 3.1, $\mathtt{R}_w$ alone does not suffice to yield an UBA. The construction from Section 3.1, for example, validates that all rejecting SCCs can be left using breakpoints, and we have shown how that leaves leeway w.r.t. how these breakpoints are met. In this section, we discuss a different, an unambiguous (but not finite) way to check the correctness of $\mathtt{R}_w$ by making the minimal time it takes from a state, for the given input word, to leave the rejecting SCC of $\mathcal{A}$ or $\widehat{\mathcal{A}}$ on every branch of this run DAG. For instance, in Figure 1, it is possible to select different successors for state q when reading a b, going to either s or t. One of them will lead to leaving this SCC immediately, either s (when reading a b) or t (when reading an a). For acceptance, the choice does not matter – so long as the correct choice is eventually made. On the word b^ω, for example in $\mathcal{A}$, we could go to t the first 20 times, and to s only in the 21^{st} attempt; the answer to the question 'how long does it take to leave the SCC starting in q on this run DAG?' would be 42.

The *shortest* time, however, is well defined. In the example automaton $\mathcal{A}$, it depends on the next letter: if it is a, then the distance is 1 from t, 2 from q, and 3 from s, and when it is b, then the distance is 1 from s, 2 from q, and 3 from t.

To reason about the minimal number of steps it takes from a state within a rejecting SCC that needs to leave it, we will define a *distance function.*

Formally, we denote by R the set of states in all rejecting SCCs of $\mathcal{A}$ and A the set of states in all accepting SCCs of $\mathcal{A}$. For a given word w and its unique sequence $\mathtt{R}_w$, we identify the unique distance[4] to leave a rejecting SCCs at each level i in $\mathcal{G}_w$ by defining a distance function $d_i : (Q_1^i \cap R) \uplus (A \setminus Q_1^i) \to \mathbb{N}^{>0}$ for each $i \in \mathbb{N}$.

▶ **Definition 5.** *Let w be a word and $\mathtt{R}_w = Q_1^0 Q_1^1 \cdots$ be its unique sequence of sets of states. We say $\Phi_w = (Q_1^0, d_0)(Q_1^1, d_1) \cdots$ is* consistent *if, for every $i \in \mathbb{N}$, we have (Q_1^i, d_i) and (Q_1^{i+1}, d_{i+1}) satisfy the following rules:*

R1. *For every state $p \in R \cap Q_1^i$ that belongs to a rejecting SCC C in $\mathcal{A}$,*

$$a: \ (Q_1^{i+1} \setminus C) \cup \{q \in C \cap Q_1^{i+1} \mid d_{i+1}(q) \leq d_i(p) - 1\} \models \delta(p, w[i]) \ \textit{and}$$

$$b: \ \textit{if } d_i(p) > 1, (Q_1^{i+1} \setminus C) \cup \{q \in C \cap Q_1^{i+1} \mid d_{i+1}(q) \leq d_i(p) - 2\} \not\models \delta(p, w[i]) \ \textit{hold.}$$

R2. *For every state $p \in A \setminus Q_1^i$ that belongs to an accepting SCC C in $\mathcal{A}$,*

$$a: \ \big(Q \setminus (Q_1^{i+1} \cup C)\big) \cup \{q \in C \setminus Q_1^{i+1} \mid d_{i+1}(q) \leq d_i(p) - 1\} \models \widehat{\delta}(p, w[i]) \ \textit{and}$$

$$b: \ \textit{if } d_i(q) > 1, \big(Q \setminus (Q_1^{i+1} \cup C)\big) \cup \{q \in C \setminus Q_1^{i+1} \mid d_{i+1}(q) \leq d_i(p) - 2\} \not\models \widehat{\delta}(p, w[i]) \ \textit{hold.}$$

Intuitively, the distance function defines a *minimal* number of steps to escape from rejecting SCCs over different accepting run DAGs and *maximal* over different branches of one such run DAG.

For instance, when $d_i(p) = 1$, we have that $Q_1^{i+1} \setminus C \models \delta(p, w[i])$ if $p \in Q_1^i \cap R$, otherwise $Q \setminus (Q_1^{i+1} \cup C) \models \widehat{\delta}(p, w[i])$ if $p \in A \setminus Q_1^i$. It means that p can escape from C within one step from an accepting run DAG $\mathcal{G}_{w[i\cdots]}$ starting from $\langle p, 0 \rangle$.

▶ **Lemma 6.** *For each $w \in \Sigma^\omega$, there is a unique consistent sequence $\Phi_w = (Q_1^0, d_0)(Q_1^1, d_2) \cdots$ where $Q_1^0 Q_1^1 Q_1^2 \cdots$ is $\mathtt{R}_w$ and $d_0 d_1 \cdots$ is the sequence of distance functions.*

One can easily construct a consistent sequence of distance functions as follows. Let C be a rejecting SCC of $\mathcal{A}$; the case for a rejecting SCC of $\widehat{\mathcal{A}}$ is entirely similar. Below, we describe how to obtain a sequence of distance values for each state $q \in C \cap Q_1^i$ with $i \geq 0$ in order to

[4] Note that, while the distance is unique, the way does not have to be. To see this, we could just expand the alphabet of $\mathcal{A}$ by adding a letter c, and by adding c to the transitions from both s and t to r. Then there are two equally short (length 2) ways from q to r whenever the next letter is c.

form a consistent sequence Φ_w. For $q \in C \cap Q_1^i$ at the level i, we first obtain an accepting run DAG $\mathcal{G}_{w[i\cdots]}$ over $w[i\cdots]$ starting from $\langle q, 0\rangle$. One can define the maximal distance, say K, over *all* branches from $\langle q, 0\rangle$ to escape the rejecting SCC C. Such a maximal distance value must exist and be a finite value, since all branches will eventually get trapped in accepting SCCs. For all accepting run DAGs $\mathcal{G}'_{w[i\cdots]}$ over $w[i\cdots]$ starting from the vertex $\langle q, 0\rangle$, there are only finitely many run DAGs of depth K from $\langle q, 0\rangle$; we denote the finite set of such run DAGs of depth K by $P_{q,i}$. We then denote the maximal distance over one *finite* run DAG $G_{q,i,K} \in P_{q,i}$ by $K_{G_{q,i,K}}$. (Note that we set the distance to ∞ for a finite branch in $G_{q,i,K}$ if it does not visit a state outside C.) We then set $d_i(q) = \min\{K_{G_{q,i,K}} : G_{q,i,K} \in P_{q,i}\} \leq K$. One of $G_{q,i,K}$ must provide the *minimal* value, so that $d_i(q)$ is well defined. This way, we can define the sequence of distance functions $\mathbf{d} = d_0 d_1 \cdots$ for the sequence $\mathtt{R}_w$. We can also prove that the sequence $\mathtt{R}_w \times \mathbf{d}$ is consistent by an induction on all the distance values $k > 0$; We refer to [19] for the details.

The proof for the uniqueness of $\mathbf{d}$ to $\mathtt{R}_w$ can also be obtained by an induction on the distance value $k > 0$; See [19] for details. The intuition is that every consistent sequence of distance functions $\mathbf{c}$ does not have smaller distance values than $\mathbf{d}$ for every state $q \in C \cap Q_1^i$ (see the construction of $\mathbf{d}$ above), and if $\mathbf{c}$ does have greater distance values for some state, a violation of the consistency constraints in Definition 5 will be found, leading to contradiction.

3.4 Unique total preorders

The range of the sequence $\mathbf{d} = d_0 d_1 d_2 \ldots$ of distance functions for $\mathtt{R}_w$ is not a priori bounded by any given *finite* number when ranging over all infinite words. Therefore, we may need *infinite* amount of memory to store $\mathtt{d}$. To allow for an abstraction of $\mathtt{d}$ that preserves uniqueness and needs only finite memory, we will abstract the values of each function d_i as families of total *preorders*, $\{\preceq_C^i\}_{C \in \mathcal{S}}$, where $\mathcal{S}$ denotes the set of SCCs in the graph of $\mathcal{A}$. For a given SCC $C \in \mathcal{S}$, a total preorder $\preceq_C^i$ is a relation defined over $H^i \times H^i$, where $H^i = C \cap Q_1^i$ if $C \subseteq R$ or $H^i = C \setminus Q_1^i$ if $C \subseteq A$; As usual, $\preceq_C^i$ is *reflexive* (i.e., for each $q \in H^i$, $q \preceq_C^i q$) and *transitive* (i.e., for each $q, r, s \in H^i$, $q \preceq_C^i r$ and $r \preceq_C^i s$ implies $q \preceq_C^i s$). We also have $q \prec_C^i r$ whenever $q \preceq_C^i r$ but $r \not\preceq_C^i q$. We write $q \simeq_C^i r$ if we have $q \preceq_C^i r$ and $r \preceq_C^i q$. Since $\preceq_C^i$ is total, for every two states $p, q \in H^i$, we have $p \preceq_C^i q$ or $q \preceq_C^i p$. Note that $\prec_C^i$ is acyclic: it is impossible for two states $q, p \in H^i$ satisfying $p \prec_C^i q$ and $q \prec_C^i p$.

Formally, we define a consistent sequence of total preorders as below.

▶ **Definition 7.** *Let $w \in \Sigma^\omega$ and $\mathtt{R}_w = Q_1^0 Q_1^1 \cdots$ be its unique sequence of sets of states. We say $\mathcal{P}_w = (Q_1^0, \{\preceq_C^0\}_{C \in \mathcal{S}})(Q_1^1, \{\preceq_C^1\}_{C \in \mathcal{S}}) \cdots$ is* consistent *if, for every $i \in \mathbb{N}$, we have that $(Q_1^i, \{\preceq_C^i\}_{C \in \mathcal{S}})$ and $(Q_1^{i+1}, \{\preceq_C^{i+1}\}_{C \in \mathcal{S}})$ satisfy the following rules:*

R1'. *$\forall q, q' \in C \cap Q_1^i \subseteq R$, we have that $q \prec_C^i q'$ iff there exists $r \in C \cap Q_1^{i+1}$ such that*

$$a: \ \{r' \in C \cap Q_1^{i+1} \mid r' \prec_C^{i+1} r\} \cup (Q_1^{i+1} \setminus C) \models \delta(q, w[i]) \text{ and}$$

$$b: \ \{r' \in C \cap Q_1^{i+1} \mid r' \prec_C^{i+1} r\} \cup (Q_1^{i+1} \setminus C) \not\models \delta(q', w[i]) \text{ hold,}$$

where $C \subseteq R$ is a rejecting SCC of $\mathcal{A}$.

R2'. *$\forall q, q' \in C \setminus Q_1^i \subseteq A$, we have $q \prec_C^i q'$ iff there exists $r \in C \setminus Q_1^{i+1}$ such that*

$$a: \ \{r' \in C \setminus Q_1^{i+1} \mid r' \prec_C^{i+1} r\} \cup \big(Q \setminus (Q_1^{i+1} \cup C)\big) \models \widehat{\delta}(q, w[i]) \text{ and}$$

$$b: \ \{r' \in C \setminus Q_1^{i+1} \mid r' \prec_C^{i+1} r\} \cup \big(Q \setminus (Q_1^{i+1} \cup C)\big) \not\models \widehat{\delta}(q', w[i]) \text{ hold,}$$

where $C \subseteq A$ is an accepting SCC of $\mathcal{A}$.

As the names indicate, the Rules R1' and R2' correspond to Rules R1 and R2, respectively, from Definition 5. We will first show that there is a consistent sequence of total preorders for each word.

▶ **Lemma 8.** *For each word $w \in \Sigma^\omega$, there exists a consistent sequence $\mathcal{P}_w = (Q_1^0, \{\preceq_C^0\}_{C\in\mathcal{S}})(Q_1^1, \{\preceq_C^1\}_{C\in\mathcal{S}})\cdots$, where $Q_1^0 Q_1^1\cdots$ is the unique sequence $\mathtt{R}_w$.*

Proof. It is simple to derive a consistent sequence $\mathcal{P}_w = (Q_1^0, \{\preceq_C^0\}_{C\in\mathcal{S}})(Q_1^1, \{\preceq_C^1\}_{C\in\mathcal{S}})\cdots$ from $\Phi_w = (Q_1^0, d_0)(Q_1^1, d_1)\cdots$ as given in Lemma 6: We can simply select, for all $i \in \mathbb{N}$ and $C \in \mathcal{S}$, $\preceq_C^i$ is the total preorder over $C \cap Q_1^i$ (if $C \subseteq R$) or $C \setminus Q_1^i$ (if $C \subseteq A$) with $p \preceq_C^i q$ iff $d_i(p) \leq d_i(q)$. In particular, $p \prec_C^i q$ iff $d_i(p) < d_i(q)$.

It is easy to verify that the sequence $\mathcal{P}_w$ as defined above is indeed consistent. For instance, for all $q, q' \in C \cap Q_1^i \subseteq R$, if $q \prec_C^i q'$, then $d_i(q) < d_i(q')$ by definition. Then we can choose the r-state in Definition 7 (Rule R1') such that $d_{i+1}(r) = d_i(q') - 1$. (Note that some such a state r must exist since $d_i(q') > d_i(q) \geq 1$.)

Combining Definition 5 (R1) and Definition 7 (R1'), we have that Rule R1b now entails R1'b, and Rule R1a entails R1'a, because $\{r' \in C \cap Q_1^{i+1} \mid r' \prec_C^{i+1} r\} \supseteq \{r' \in C \cap Q_1^{i+1} \mid d_{i+1}(r') \leq d_i(q) - 1\}$, because $d_i(q) - 1 \leq d_i(q') - 2 < d_i(q') - 1 = d_{i+1}(r)$.

The argument for accepting SCCs is using rules R2 and R2' in the same way. ◀

After discussing how such a sequence can be obtained, we now establish that it is unique. Note, however, that it is unique for the correct sequence $\mathtt{R}_w$, while there may be sequences of total preorders that work with incorrect sequences of sets of states. (For example, a total preorder can accommodate an infinite distance for all states, where the obligation to leave a rejecting SCC cannot be discharged, while the local consistency constraints can be met.) Nonetheless, a breakpoint construction ensures to obtain the unique sequence $\mathtt{R}_w$.

▶ **Lemma 9.** *Let w be a word in Σ^ω and $\Phi_w = (Q_1^0, d_0)(Q_1^1, d_1)\cdots$ be its unique consistent sequence of distance functions. Let $\mathcal{P}_w = (Q_1^0, \{\preceq_C^0\}_{C\in\mathcal{S}})(Q_1^1, \{\preceq_C^1\}_{C\in\mathcal{S}})\cdots$ be a sequence satisfying Definition 7. Then*

- *For every two states $q, q' \in C \cap Q_1^i \subseteq R$, if $q \preceq_C^i q'$, then $d_i(q) \leq d_i(q')$, and in particular if $q \prec_C^i q'$, then $d_i(q) < d_i(q')$. (C is a rejecting SCC)*
- *For every two states $q, q' \in C \setminus Q_1^i \subseteq A$, if $q \preceq_C^i q'$, then $d_i(q) \leq d_i(q')$, and in particular if $q \prec_C^i q'$, then $d_i(q) < d_i(q')$. (C is an accpting SCC)*

Proof. We only prove the first claim; the proof of the second claim is entirely similar.

Let C be a rejecting SCC and i be a natural number. We let q and q' be two states in $C \cap Q_1^i$. In order to prove that $q \preceq_C^i q'$ implies $d_i(q) \leq d_i(q')$, we can just prove its contraposition that $d_i(q') < d_i(q)$ implies $q' \prec_C^i q$ for all distance values $k > 0$ with $d_i(q') \leq k$. We can similarly prove that $q \prec_C^i q'$ implies $d_i(q) < d_i(q')$.

Our goal is then to prove that, for all $k > 0$, $d_i(q') < d_i(q) \implies q' \prec_C^i q$ and $d_i(q') \leq d_i(q) \implies q' \preceq_C^i q$ when $d_i(q') \leq k$. In the remainder of the proof, we will prove it by induction over the distance value $k > 0$. Note that our claim is quantified over all natural numbers i.

For the **induction basis** ($k = 1$), we have $d_i(q') \leq k$ by assumption. So, $d_i(q') = 1$. But then $Q_1^{i+1} \setminus C \models \delta(q', w[i])$. Consequently, by Rule R1'b, q' must be a minimal element of $\preceq_C^i$. Hence, we have $q' \preceq_C^i q$. Since by assumption that $d_i(q) > d_i(q') = 1$, Rule R1 supplies $Q_1^{i+1} \setminus C \not\models \delta(q, w[i])$. We can therefore choose r from Rule R1' as a minimal element of $\preceq_C^{i+1}$ to get $S^{i+1} = \{\, r' \in C \cap Q_1^{i+1} \mid r' \prec_C^{i+1} r \,\} = \emptyset$. It follows that $S^{i+1} \cup (Q_1^{i+1} \setminus C) \models \delta(q', w[i])$ (R1'a) but $S^{i+1} \cup (Q_1^{i+1} \setminus C) \not\models \delta(q, w[i])$ (R1'b). By Definition 7, we have $q' \prec_C^i q$. Hence, for $k = 1$ and $d_i(q') \leq k = 1$, it holds that $d_i(q') < d_i(q)$ implies $q' \prec_C^i q$.

When $d_i(q) = d_i(q') = k = 1$, it directly follows that $q \not\prec^i_C q'$ and $q' \not\prec^i_C q$ by Definition 7, thus also $q' \simeq^i_C q$ since $\preceq^i_C$ is a total preorder. Therefore, if $d_i(q') \leq d_i(q)$, then $q' \preceq^i_C q$, thus also $q \prec^i_C q'$ implies $d_i(q) < d_i(q')$.

For the **induction step** $k \mapsto k+1$, we have $d_i(q') = k+1$ and we want to prove $q' \prec^i_C q$ when $k+1 = d_i(q') < d_i(q)$, and prove $q' \simeq^i_C q$ when $d_i(q') = d_i(q)$ (hence $d_i(q') \leq d_i(q) \implies q' \preceq^i_C q$). We only give the high level proof idea here and refer to [19] for details.

Recall that in the induction basis, we proved that q' is a minimal element with respect to $\preceq^i_C$ when $d_i(q') \leq k$. Our key observation is that, for all $k > 0$, all elements in $\{\, p \in C \cap Q^i_1 \mid d_i(p) = k+1 \,\}$ are minimal with respect to $\preceq^i_C$ in the set $\{\, p \in C \cap Q^i_1 \mid d_i(p) > k \,\}$ (See [19] for proof details). The intuition is that our claim is equivalent to that for every two states $q, q' \in C \cap Q^i_1 \subseteq R$, $q \preceq^i_C q'$ if and only if $d_i(q) \leq d_i(q')$ (Since $\preceq^i_C$ is a preorder, we also have $q \prec^i_C q'$ iff $d_i(q) < d_i(q')$). Hence, the minimal elements in $\{\, p \in C \cap Q^i_1 \mid d_i(p) > k \,\}$ (i.e., $\{\, p \in C \cap Q^i_1 \mid d_i(p) = k+1 \,\}$) must also be the minimal elements with respect to $\preceq^i_C$, based on our induction hypothesis.

Let $S = \{p \in C \cap Q^i_1 \mid d_i(p) > k\}$. First, we know that q' is a minimal element with respect to $\preceq^i_C$ in the set S, as $d_i(q') = k+1$ by assumption. Since by assumption that $k < d_i(q') = k+1 < d_i(q)$, we know that q is also in S. Hence, $q' \preceq^i_C q$ holds.

We still need to prove that $q' \prec^i_C q$ under the assumption that $d_i(q') < d_i(q)$. By assumption that $d_i(q) > d_i(q') = k+1$, we pick a state r' that is minimal w.r.t. $\preceq^{i+1}_C$ in the set $\{p \in C \cap Q^{i+1}_1 \mid d_{i+1}(p) > k\}$ (and hence $d_{i+1}(r') = k+1$). We then prove that the selected state r' is the r-state that witnesses $q' \prec^i_C q$ for R1' of Definition 7. The observation is that, by Definition 5, we have $Q^{i+1}_1 \setminus C \cup \{\, p \in C \cap Q^{i+1}_1 \mid d_{i+1}(p) \leq d_i(q') - 1 = d_{i+1}(r') - 1 \,\} \models \delta(q', w[i])$ but $Q^{i+1}_1 \setminus C \cup \{\, p \in C \cap Q^{i+1}_1 \mid d_{i+1}(p) \leq d_{i+1}(r') - 1 \,\} \not\models \delta(q, w[i])$. By induction hypothesis, for all states $p \in C \cap Q^{i+1}_1$ such that $d_{i+1}(p) \leq d_{i+1}(r') - 1 = k$ (i.e., $d_{i+1}(p) < d_{i+1}(r')$), we also have $p \prec^i_C r'$. It then follows that by Definition 7 that $q' \prec^i_C q$ holds. Hence, $d_i(q') < d_i(q) \implies q' \prec^i_C q$.

To prove that $q \prec^i_C q'$ implies $d_i(q) < d_i(q')$, we also prove its contraposition, i.e., $d_i(q') \leq d_i(q)$ implies $q' \preceq^i_C q$ for all $i \in \mathbb{N}$. We have already shown that $d_i(q') < d_i(q)$ implies $q' \prec^i_C q$. Moreover, if $d_i(q') = d_i(q) = k+1$, then $q' \simeq^i_C q$, since both q' and q are minimal element w.r.t. $\preceq^i_C$ in the set $\{p \in C \cap Q^i_1 \mid d_i(p) > k\}$. It then follows that $q \prec^i_C q'$ implies $d_i(q) < d_i(q')$. Hence, we have completed the proof. ◀

By Lemma 9, for states $p, q \in H^i$, we have both $p \simeq^i_C q \iff d_i(p) = d_i(q)$ and $p \prec^i_C q \iff d_i(p) < d_i(q)$ hold for all $i \in \mathbb{N}$, where $H^i = C \cap Q^i_1$ if $C \subseteq R$ and $H^i = C \setminus Q^i_1$ if $C \subseteq A$. Then Corollary 10 follows immediately from Lemma 6.

▶ **Corollary 10.** *For each $w \in \Sigma^\omega$, there is a unique consistent sequence of sets of states and total preorders $\mathcal{P}_w = (Q^0_1, \{\preceq^0_C\}_{C \in \mathcal{S}})(Q^1_1, \{\preceq^1_C\}_{C \in \mathcal{S}}) \cdots$ where $Q^0_1 Q^1_1 Q^2_1 \cdots$ is the unique sequence $\mathtt{R}_w$.*

In order to lift this unique set to an UBA, we need to discharge the correctness of the sequence $Q^0_1 Q^1_1 Q^2_1 \cdots$. This is, however, a relatively simple task: for the correct sequence, the total preorders provide the same rational way of creating the same accepting runs on the tails $w[i \cdots]$ of w from the states marked as accepting in $\mathcal{A}$ by inclusion in Q^i_1, or as accepting from $\widehat{\mathcal{A}}$ by non-inclusion in Q^i_1.

To prepare such a construction, we first define an arbitrary (but fixed) order on the SCCs of $\mathcal{A}$, as well as a $\mathtt{next}$ operator for cycling through SCCs, and fix an initial SCC $C_0 \in \mathcal{S}$. Recall that $\mathcal{S}$ is the set of all SCCs in $\mathcal{A}$. Note that we assume that the graph of $\mathcal{A}$ has at least one SCC. If this is not the case, we can simply build an unambiguous safety automaton that guesses $\mathtt{R}_w$. Then, our construction of UBA is formalized below.

▶ **Definition 11.** *Let $\mathcal{A} = (\Sigma, Q, \iota, \delta, F)$ be an AWA. We define an NBA $\mathcal{B}_u = (\Sigma, Q_u, I_u, \delta_u, F_u)$ as follows.*

- *The macrostates of Q_u are tuples $(Q_1, Q_2, \{\preceq_C\}_{C\in\mathcal{S}}, S, D)$ such that*
  - *Q_1 and Q_2 partition Q, i.e., $Q_2 = Q \setminus Q_1$*
  - *for all $C \in \mathcal{S}$, if $C \subseteq R$ then $\preceq_C$ is a total preorder over $Q_1 \cap C$*
  - *for all $C \in \mathcal{S}$, if $C \subseteq A$ then $\preceq_C$ is a total preorder over $Q_2 \cap C$*
  - *$S \in \mathcal{S}$ is an SCC in the graph of $\mathcal{A}$*
  - *D is a downwards closed set w.r.t. the total preorder $\preceq_S$: if $q \in D$ then (1) $q \in Q_1 \cap S$ if $S \subseteq R$ resp. $q \in Q_2 \cap S$ if $S \subseteq A$, and (2) $q' \preceq_S q$ implies $q' \in D$,*
- *$I_u = \{\, (Q_1, Q_2, \{\preceq_C\}_{C\in\mathcal{S}}, S, D) \in Q_u \mid \iota \in Q_1, S = C_0, D = \emptyset \,\}$,*
- *Let $(Q_1, Q_2, \{\preceq_C\}_{C\in\mathcal{S}}, S, D)$ be a macrostate in Q_u and $\sigma \in \Sigma$. Then we have that $(Q'_1, Q'_2, \{\preceq'_C\}_{C\in\mathcal{S}}, S', D') \in \delta_u\big((Q_1, Q_2, \{\preceq_C\}_{C\in\mathcal{S}}, S, D), \sigma\big)$ if*
  - *$Q'_1 \models \wedge_{s\in Q_1}\delta(s, \sigma)$ and $Q'_2 \models \wedge_{s\in Q_2}\widehat{\delta}(s, \sigma)$ (local consistency)*
  - *for all $C \in \mathcal{S}$, $(Q_1, \preceq_C)$ and $(Q'_1, \preceq'_C)$ satisfy the requirements of Rule R1' (if $C \subseteq R$) resp. Rule R2' (if $C \subseteq A$)*
  - *if $D = \emptyset$, then $S' = \mathsf{next}(S)$ and $D' = Q'_1 \cap S'$ if $S' \subseteq R$ resp. $D' = Q'_2 \cap S'$ if $S' \subseteq A$,*
  - *if $D \neq \emptyset$, then $S' = S$ and D' is the smallest downwards closed set (see above) such that $D' \cup (Q'_1 \setminus S) \models \wedge_{s\in D}\delta(s, \sigma)$ if $S \subseteq R$ resp. $D' \cup (Q'_2 \setminus S) \models \wedge_{s\in D}\widehat{\delta}(s, \sigma)$ if $S \subseteq A$,*
- *$F_u = \{\, (Q_1, Q_2, \{\preceq_C\}_{C\in\mathcal{S}}, S, D) \in Q_u \mid D = \emptyset \,\}$.*

The new construction uses D as the breakpoint to ensure that the correct unique sequence $\mathtt{R}_w$ for each word w is obtained. The nondeterminism of the construction lies only in choosing Q'_1 (which entails Q'_2) and in updating the total preorders. From an accepting macrorun of $\mathcal{B}_u$ over a word w, one can actually construct an accepting run DAG $\mathcal{G}_w$ of $\mathcal{A}$ by selecting successors in the next level for each state q only the ones in the smallest downwards closed set D satisfying $\delta(q, \sigma)$. This way, all branches of $\mathcal{G}_w$ by construction will eventually get trapped in an accepting SCC, since D will become empty infinitely often. Hence, $\mathcal{L}(\mathcal{B}_u) \subseteq \mathcal{L}(\mathcal{A})$. Moreover, one can construct from the unique sequence of preorders Φ_w of a word $w \in \mathcal{L}(\mathcal{A})$ as given in Corollary 10 a unique infinite macrorun ρ of $\mathcal{B}_u$. Wrong guesses of the preorders for $\mathtt{R}_w$ will result in discontinued macroruns once a violation to R1' (or R2') has been detected. That is, there are no consistent ways to update the preorders in the next macrostate. Further, by Lemma 9, we have that $d_i(q) = d_i(q') \Leftrightarrow q \simeq^i_C q'$ and $d_i(q) < d_i(q') \Leftrightarrow q \prec^i_C q'$ for all $i \in \mathbb{N}$. So, by Definition 5 and Definition 7, one can observe that, if $D^i \neq \emptyset$, $\sup\{d_i(q) \mid q \in D^i\} = \sup\{d_{i+1}(q) \mid q \in D^{i+1}\} + 1$ (choosing $\sup\emptyset = 0$), where D^i is the D-component of macrostate $\rho[i]$ with $i \in \mathbb{N}$. Since for every nonempty D^i, $\sup\{d_i(q) \mid q \in D^i\}$ is finite and the maximal value in D^i is always decreasing, the value will eventually become 0, i.e., D always becomes empty eventually. That is, ρ must be accepting. Hence, Theorem 12 follows; See [19] for more details.

▶ **Theorem 12.** *Let $\mathcal{B}_u$ be defined as in Definition 11. Then (1) $\mathcal{L}(\mathcal{B}_u) = \mathcal{L}(\mathcal{A})$, and (2) $\mathcal{B}_u$ is unambiguous.*

▶ **Example 13.** Consider again the AWW $\mathcal{A}$ depicted in Figure 1. Recall that, in Figure 1, the macrostate $(Q, \{q, s, t\})$ has two successors over b because of the nondeterminism in developing breakpoints. We now apply Definition 11 to construct a UBA $\mathcal{B}_u$ from $\mathcal{A}$. There are three SCCs in $\mathcal{A}$, namely $C_0 = \{p\}, C_1 = \{q, s, t\}$ and $C_2 = \{r\}$. Since C_0 and C_2 both have only one state, the total preorders for them are fixed and thus ignored here. We only need to guess the preorder over C_1. Let us consider the constucted $\mathcal{B}_u$ over b^ω starting from the macrostate $m_0 = (Q, \{\}, \preceq^0_{C_1}, C_1, C_1)$ where $\preceq^0_{C_1}$ is defined as $\{s \prec^0_{C_1} q \prec^0_{C_1} t\}$.

First, recall that $\mathtt{R}_{b^\omega} = Q^\omega$. Obviously, $m_{1a} = (Q, \{\}, \{s \prec^1_{C_1} q \prec^1_{C_1} t\}, C_1, \{q, s\})$, which corresponds to $(Q, \{q, s\})$ in Figure 1, is a valid successor of m_0 over b, while $m_{1b} = (Q, \{\}, \{s \prec^1_{C_1} q \prec^1_{C_1} t\}, C_1, \{q, t\})$, which corresponds to $(Q, \{q, t\})$ in Figure 1, is not. The reason is that $\{q, t\}$ is *not* a downwards closed set with respect to $\preceq^1_{C_1}$, since we have $s \prec^1_{C_1} t$ but s is missing in the breakpoint set. One may wonder whether we can change the preorder $\preceq^1_{C_1}$ and consider the candidate successor $m_{1c} = (Q, \{\}, \{q \prec^2_{C_1} t \prec^2_{C_1} s\}, \{q, t\})$. Indeed, $\{q, t\}$ is now a downwards closed set with respect to $\preceq^2_{C_1}$. However, $(Q, \preceq^0_{C_1})$ and $(Q, \preceq^2_{C_1})$ do not satisfy the local consistency as required by Definition 7. First, we have that $Q \setminus C_1 \cup \{\} \models \delta(s, b)$. So, there do not exist r-states in $C_1 \cap Q$ that witness $q \prec^2_{C_1} s$ and $t \prec^2_{C_1} s$, as required by R1' of Definition 7. In fact, one can verify that $s \prec_{C_1} q \prec_{C_1} t$ is the only valid preorder over C_1 when the input word is b^ω. This is due to the fact that when reading b, the distance to escape C_1 is 1 from s, 2 from q, and 3 from t. Hence, m_{1c} must not be a valid successor of m_0. The accepting macrorun of $\mathcal{B}_u$ (from Definition 11) over b^ω is $(Q, \{\}, \{s \prec_{C_1} q \prec_{C_1} t\}, C_0, \{\}) \xrightarrow{b} (Q, \{\}, \{s \prec_{C_1} q \prec_{C_1} t\}, C_1, \{q, s, t\}) \xrightarrow{b} (Q, \{\}, \{s \prec_{C_1} q \prec_{C_1} t\}, C_1, \{q, s\}) \xrightarrow{b} (Q, \{\}, \{s \prec_{C_1} q \prec_{C_1} t\}, C_1, \{s\}) \xrightarrow{b} (Q, \{\}, \{s \prec_{C_1} q \prec_{C_1} t\}, C_1, \{\}) \xrightarrow{b} (Q, \{\}, \{s \prec_{C_1} q \prec_{C_1} t\}, C_2, \{\}) \xrightarrow{b} (Q, \{\}, \{s \prec_{C_1} q \prec_{C_1} t\}, C_0, \{\}) \cdots$.

4 Improvements and Complexity

When revisiting the construction in search for improvements, it seems wasteful to keep total preorders for all SCCs in the graph of $\mathcal{A}$, given that they are not interacting with each other. Can we focus on just one at a time? It proves to be possible to optimise the automaton from Definition 11 in this way, with re-establishing uniqueness proving the greatest obstacle. The resulting automaton is smaller in practice, mainly because it only keeps track of a total preorder over only one SCC.

We provide this construction only as an improvement over the principle construction from Definition 11 for two reasons. First, while this provides quite a significant advantage where there are many small SCCs rather than one big SCC, this has little effect on the worst case (which occurs when there is one SCC, cf. Theorem 16). Second, it loosens the connection that the total preorders from Definition 11 need to be the natural abstraction of the unique distance function from Definition 5.

▶ **Definition 14.** *Let $\mathcal{A} = (\Sigma, Q, \iota, \delta, F)$ be an AWA. We define an NBA $\mathcal{U} = (\Sigma, Q_u, I_u, \delta_u, F_u)$ as follows.*

- *The macrostates of Q_u are tuples $(Q_1, Q_2, \preceq_C, C, D)$ such that*
  - *Q_1 and Q_2 partition Q*
  - *C is an SCC in the graph of $\mathcal{A}$ and*
    - ∗ *if $C \subseteq R$ then $\preceq_C$ is a total preorder of $Q_1 \cap C$*
    - ∗ *if $C \subseteq A$ then $\preceq_C$ is a total preorder of $Q_2 \cap C$*
  - *let M be the set of maximal elements of the total preorder $\preceq_C$, and let $H = C \cap Q_1$ if $C \subseteq R$ resp. $H = C \cap Q_2$ if $C \subseteq A$; then $D = H$ or $D = H \setminus M$*
- *$I_u = \{\, (Q_1, Q_2, \preceq_C, C, D) \in Q_u \mid \iota \in Q_1, C = C_0, D = \emptyset \,\}$,*
- *Let $(Q_1, Q_2, \preceq_C, C, D)$ be a macrostate in Q_u and $\sigma \in \Sigma$. Then we have that $(Q'_1, Q'_2, \preceq'_{C'}, C', D') \in \delta_u\big((Q_1, Q_2, \preceq_C, C, D), \sigma\big)$ if*
  - *$Q'_1 \models \wedge_{s \in Q_1} \delta(s, \sigma)$ and $Q'_2 \models \wedge_{s \in Q_2} \widehat{\delta}(s, \sigma)$ (local consistency)*
  - *if $D = \emptyset$, then $C' = \mathsf{next}(C)$ and $D' = Q'_1 \cap C'$ if $C' \subseteq R$ resp. $D' = Q'_2 \cap C'$ if $C' \subseteq A$,*
  - *if $D \neq \emptyset$ then $C' = C$,*

  - ∗ *$(Q_1, \preceq_C)$ and $(Q'_1, \preceq'_C)$ must satisfy the requirements of Rule R1' (if $C \subseteq R$) resp. Rule R2' (if $C \subseteq A$) and*
  - ∗ *D' is the smallest downward closed set w.r.t. $\preceq'_C$ such that*[5] *$D' \cup (Q'_1 \setminus C) \models \wedge_{s \in D}\delta(s, \sigma)$ if $C \subseteq R$ resp. $D' \cup (Q'_2 \setminus C) \models \wedge_{s \in D}\delta(s, \sigma)$ if $C \subseteq A$,*
- $F_u = \{ (Q_1, Q_2, \preceq_C, C, D) \in Q_u \mid D = \emptyset \}$.

The nondeterminism of the construction again lies in choosing Q'_1 (which entails Q'_2) and in updating the total preorder. One can also construct from an accepting macrorun of $\mathcal{U}$ over w an accepting run DAG $\mathcal{G}_w$ of $\mathcal{A}$, using the same way as we did for Theorem 12. So, $\mathcal{L}(\mathcal{U}) \subseteq \mathcal{L}(\mathcal{A})$.

For the other direction, we first observe that the preorders of *every* accepting macrorun $(Q^0_1, Q^0_2, \preceq_0, S^0, D^0)(Q^1_1, Q^1_2, \preceq_1, S^1, D^1)\cdots$ of $\mathcal{U}$ over w can be tightly related with the distance values of states defined in $\mathbf{d}$. More precisely, let $D^{i'} = D^i = \emptyset$ with $i' < i$ being two consecutive accepting positions. Then for all $j \in (i', i]$, we have that:

1. for all $q \in D^j$ and all $q' \in C^i \cap Q^j_1$. $d_j(q) \leq d_j(q') \Leftrightarrow q \preceq_j q'$, and $d_j(q) \leq i - j$ hold,
2. for all $q \in C^i \cap Q^j_1$ and all $q' \in M^j = (C^i \cap Q^j_1) \setminus D^j$. $q \preceq_j q'$ and $d_j(q') > i - j$ hold, and
3. $m_j = \sup\{d_j(q) \mid q \in D^j\} = i - j$, using $\sup \emptyset = 0$,

where $C^i \subseteq R$ is a rejecting SCC of $\mathcal{A}$. Note that $C^j = C^i$ for all $i' < j \leq i$. The case for $C^i \subseteq A$ can be defined similarly. Let $m_j = \sup\{d_j(q) \mid q \in D^j\}$. The intuition is that all states in $M^j = (C^i \cap Q^j_1) \setminus D^j = \{ s \in C^i \cap Q^j_1 \mid d_j(s) > m_i \}$ are aggregated by construction as the maximal elements w.r.t. $\preceq_j$, while $\preceq_j$ orders all states in $D^j = \{ s \in C^i \cap Q^j_1 \mid d_j(s) \leq m_j \}$ exactly as in the preorders of Corollary 10. So, the correspondence between d_j and $\preceq_j$ in the three items then follows naturally. For technical reasons, if $q \in D^j$ or $q' \in (C^i \cap Q^j_1) \setminus D^j$ do not exist in above items, we say the item above still holds. See [19] for proof details.

In fact, one can construct such an accepting macrorun satisfying the three items above for $\mathcal{U}$ by simulating $\mathcal{B}_u$ as follows. If $\rho = (Q^0_1, Q^0_2, \{\preceq^0_C\}_{C \in \mathcal{S}}, S^0, D^0)(Q^1_1, Q^1_2, \{\preceq^1_C\}_{C \in \mathcal{S}}, S^1, D^1)(Q^2_1, Q^2_2, \{\preceq^2_C\}_{C \in \mathcal{S}}, S^2, D^2)\cdots$ is the accepting macrorun of $\mathcal{B}_u$ on a word w, then $\mathcal{U}$ has an accepting macrorun $\widehat{\rho} = (Q^0_1, Q^0_2, \preceq_0, S^0, D^0)(Q^1_1, Q^1_2, \preceq_1, S^1, D^1)(Q^2_1, Q^2_2, \preceq_2, S^2, D^2)\cdots$ (that differs from ρ only in preorders), such that

- if $S^i \subseteq R$, then $\preceq_i$ is a total preorder on $S^i \cap Q^i_1$ where $\preceq_i = \preceq^i_{S^i}$ if $D^i = S^i \cap Q^i_1$ and otherwise, the maximal elements M^i of $\preceq_i$ are $(S^i \cap Q^i_1) \setminus D^i$, and the restriction of $\preceq_i$ to $D^i \times D^i$ agrees with the restriction of $\preceq^i_{S^i}$ to $D^i \times D^i$, and
- similarly, if $S^i \subseteq A$, then $\preceq_i$ is a total preorder on $S^i \cap Q^i_2$ where $\preceq_i = \preceq^i_{S^i}$ if $D^i = S^i \cap Q^i_2$ and otherwise, the maximal elements M^i of $\preceq_i$ are $(S^i \cap Q^i_2) \setminus D^i$, and the restriction of $\preceq_i$ to $D^i \times D^i$ agrees with the restriction of $\preceq^i_{S^i}$ to $D^i \times D^i$.

It is easy to verify that $\widehat{\rho}$ satisfies all local constraints for Rule R1' resp. R2'. Hence, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B}_u) \subseteq \mathcal{L}(\mathcal{U})$, thus also $\mathcal{L}(\mathcal{U}) = \mathcal{L}(\mathcal{A})$.

One can show that $\widehat{\rho}$ is the sole accepting macrorun of $\mathcal{U}$ over w by the following facts. (i) There is only a single initial macrostate that fits $\mathtt{R}_w$, and when we take a transition from an accepting macrostate (including the first), the next SCC is deterministically selected; (ii) Moreover, all relevant states from this SCC are in the D^i component and $m_i = \sup\{d_i(q) \mid q \in D^i\}$ is the distance to the next breakpoint (by Item (3) above), and thus the $\preceq_i$ and D^i up to it. With a simple inductive argument we can thus conclude that $\widehat{\rho}$ is the only such accepting macrorun. Then, Theorem 15 follows.

[5] Note that this is a deterministic assignment that does not necessarily lead to a set D' that covers all of $\preceq'_C$ or all of $\preceq'_C$ except for the maximal elements; if it does not, then this transition is disallowed

▶ **Theorem 15.** *Let $\mathcal{U}$ be defined as in Definition 14. Then (1) $\mathcal{L}(\mathcal{U}) = \mathcal{L}(\mathcal{A})$ and (2) $\mathcal{U}$ is unambiguous.*

We now turn to the complexity of our constructions. Let $\mathsf{tpo}(n)$ denote the number of total preorders over a set with n states. By [3], $\mathsf{tpo}(n) \approx \frac{n!}{2(\ln 2)^{n+1}}$, so that we get $\lim_{n\to\infty} \frac{\sqrt[n]{\mathsf{tpo}(n)}}{n} = \lim_{n\to\infty} \frac{\sqrt[n]{n!}}{n} \cdot \frac{1}{\sqrt[n]{2\ln 2}} \cdot \frac{1}{\ln 2} = \frac{1}{e} \cdot 1 \cdot \frac{1}{\ln 2} = \frac{1}{e \ln 2} \approx 0.53$. Hence, $\mathsf{tpo}(n) \approx (0.53n)^n$, which is a better bound than the best known bound $(0.76n)^n$ for Büchi disambiguation [16] and complementation [24].

▶ **Theorem 16.** *If $\mathcal{A}$ has n states, then the numbers of states of $\mathcal{U}$ and $\mathcal{B}_u$ are $\mathcal{O}\big(\mathsf{tpo}(n)\big)$ and $\mathcal{O}\big(n \cdot \mathsf{tpo}(n)\big)$, respectively.*

Proof. For both automata, the worst case occurs when all states are in the same SCC C, say $C = R$. Starting with $\mathcal{U}$, each macrostate is a tuple $(Q_1, C \setminus Q_1, \preceq, C, D)$. There are four possibilities for the tuple, namely $C = Q_1 = D$, $C = Q_1 \supsetneq D$, $C \supsetneq Q_1 = D$, and $C \supsetneq Q_1 \supsetneq D$. For each of these four cases, we can produce an injection from the tuple (macrostate) onto a total preorder $\preceq'$ over C, so that we have at most $4 \cdot \mathsf{tpo}(n)$ macrostates. For $C = Q_1 = D$, for example, we can keep the $\preceq$ over C, i.e., we set $\preceq' = \preceq$. When there is strict inclusion, i.e., $C \supsetneq Q_1$, we extend the $\preceq$ on Q_1 to a total preorder $\preceq'$ over C by adding the states in $C \setminus Q_1$ resp. $Q_1 \setminus D$ as minimal resp. maximal elements (with their separate equivalence class). For each of the four cases, the respective mapping is injective.

As this covers all macrostates of $\mathcal{U}$, $\mathcal{U}$ has at most $4 \cdot \mathsf{tpo}(n)$ macrostates.

For $\mathcal{B}_u$, there are $\mathcal{O}(n)$ possible choices for D, since the maximal element in D with respect to the preorder $\preceq$ has at most n possibilities. This leads to $\mathcal{O}(n \cdot \mathsf{tpo}(n))$ macrostates. ◀

5 Discussion

We have given the *first* direct translation from AWAs to UBAs. The complexity of our translation is even *smaller* than that of the best known disambiguation algorithm for NBAs [16] (broadly $(0.53n)^n$ vs. $(0.76n)^n$). We can further optimise the construction of $\mathcal{U}$ slightly by moving to *transition-based* acceptance conditions. That is, an ω-word is now accepted by $\mathcal{U}$ if one of its corresponding runs visits accepting transitions for infinitely many times. Essentially, where $(Q_1', Q_2', \preceq', C, \emptyset) \in \delta_u\big((Q_1, Q_2, \preceq, C, D), \sigma\big)$, $(Q_1', Q_2', \preceq'$ $, C, \emptyset)$ would be replaced by $\delta_u\big((Q_1, Q_2, \equiv, C, \emptyset), \sigma\big)$. ($\equiv$ identifies all states it compares; it is the only total preorder acceptable for $D = \emptyset$.)

This is done recursively, until the only macrostates with $D = \emptyset$ left are those with $Q_1 \cap R = \emptyset = Q_2 \cap A$ and (arbitrarily) $C = C_0$. Note that the initial macrostate has to be changed for this, too.

Removing most macrostates with $D = \emptyset$, this reduces the statespace slightly. It is also the automaton obtained by de-generalising the standard LTL to transition-based unambiguous generalized Büchi automaton construction. We can also “re-generalise”: every singleton SCC can be removed from the round-robin at the cost of including an individual Büchi condition that accepts when the state s is not in Q_1 or Q_2, respectively, or if $Q_1 \models \delta(s, \sigma)$ or $Q_2 \models \widehat{\delta}(s, \sigma)$, respectively, holds. If all components are singleton, we obtain the standard constuction for AVAs / LTL since the preorders of our construction given in Section 4 can be omitted. This way, the D set in a macrostate degenerates to a purely breakpoint construction. Then, the improved complexity for AVAs matches the current known bounds $n2^n$ for the LTL-to-UBA construction [14, 26].

References

1 Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

2 Christel Baier, Stefan Kiefer, Joachim Klein, Sascha Klüppelholz, David Müller, and James Worrell. Markov chains and unambiguous Büchi automata. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification – 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2016. doi:10.1007/978-3-319-41528-4_2.

3 J.P. Barthelemy. An asymptotic equivalent for the number of total preorders on a finite set. *Discrete Mathematics*, 29(3):311–313, 1980. doi:10.1016/0012-365X(80)90159-4.

4 Michael Benedikt, Rastislav Lenhardt, and James Worrell. LTL model checking of interval markov chains. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2013. doi:10.1007/978-3-642-36742-7_3.

5 Frantisek Blahoudek, Juraj Major, and Jan Strejcek. LTL to smaller self-loop alternating automata and back. In Robert M. Hierons and Mohamed Mosbah, editors, *Theoretical Aspects of Computing – ICTAC 2019 – 16th International Colloquium, Hammamet, Tunisia, October 31 – November 4, 2019, Proceedings*, volume 11884 of *Lecture Notes in Computer Science*, pages 152–171. Springer, 2019. doi:10.1007/978-3-030-32505-3_10.

6 Udi Boker, Orna Kupferman, and Adin Rosenberg. Alternation removal in Büchi automata. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2010. doi:10.1007/978-3-642-14162-1_7.

7 Udi Boker, Karoliina Lehtinen, and Salomon Sickert. On the translation of automata to linear temporal logic. In Patricia Bouyer and Lutz Schröder, editors, *Foundations of Software Science and Computation Structures – 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13242 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2022. doi:10.1007/978-3-030-99253-8_8.

8 J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congress on Logic, Method, and Philosophy of Science. 1960*, pages 1–12. Stanford University Press, 1962.

9 Doron Bustan, Sasha Rubin, and Moshe Y. Vardi. Verifying omega-regular properties of Markov chains. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2004. doi:10.1007/978-3-540-27813-9_15.

10 Olivier Carton and Max Michel. Unambiguous Büchi automata. *Theor. Comput. Sci.*, 297(1-3):37–81, 2003. doi:10.1016/S0304-3975(02)00618-7.

11 Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001. doi:10.1007/3-540-44585-4_6.

12 Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860. IJCAI/AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997.

13 Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. doi:10.1109/32.588521.

14 Simon Jantsch, David Müller, Christel Baier, and Joachim Klein. From LTL to unambiguous Büchi automata via disambiguation of alternating automata. *Formal Methods Syst. Des.*, 58(1-2):42–82, 2021. doi:10.1007/s10703-021-00379-z.

15 Detlef Kähler and Thomas Wilke. Complementation, disambiguation, and determinization of Büchi automata unified. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 724–735. Springer, 2008. doi:10.1007/978-3-540-70575-8_59.

16 Hrishikesh Karmarkar, Manas Joglekar, and Supratik Chakraborty. Improved upper and lower bounds for Büchi disambiguation. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis – 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2013. doi:10.1007/978-3-319-02444-8_5.

17 Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Log.*, 2(3):408–429, 2001. doi:10.1145/377978.377993.

18 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification – 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi:10.1007/978-3-642-22110-1_47.

19 Yong Li, Sven Schewe, and Moshe Y. Vardi. Singly exponential translation of alternating weak büchi automata to unambiguous büchi automata. *CoRR*, abs/2305.09966, 2023. doi:10.48550/arXiv.2305.09966.

20 Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984. doi:10.1016/0304-3975(84)90049-5.

21 David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Alternating automata, the weak monadic theory of trees and its complexity. *Theor. Comput. Sci.*, 97(2):233–244, 1992. doi:10.1016/0304-3975(92)90076-R.

22 David E. Muller and Paul E. Schupp. Alternating automata on infinite objects, determinacy and rabin's theorem. In Maurice Nivat and Dominique Perrin, editors, *Automata on Infinite Words, Ecole de Printemps d'Informatique Théorique, Le Mont Dore, France, May 14-18, 1984*, volume 192 of *Lecture Notes in Computer Science*, pages 100–107. Springer, 1984. doi:10.1007/3-540-15641-0_27.

23 Gareth Scott Rohde. *Alternating automata and the temporal logic of ordinals.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

24 Sven Schewe. Büchi complementation made tight. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*, volume 3 of *LIPIcs*, pages 661–672. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2009. doi:10.4230/LIPIcs.STACS.2009.1854.

25 Moshe Y. Vardi. The rise and fall of LTL. In Giovanna D'Agostino and Salvatore La Torre, editors, *Proceedings of Second International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2011, Minori, Italy, 15-17th June 2011*, 2011. invited talk. URL: https://www.cs.rice.edu/~vardi/papers/gandalf11-myv.pdf.

26 Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 332–344. IEEE Computer Society, 1986.

Contextual Behavioural Metrics

Ugo Dal Lago
University of Bologna, Italy
INRIA Sophia Antipolis, France

Maurizio Murgia
Gran Sasso Science Institute, L'Aquila, Italy

Abstract

We introduce contextual behavioural metrics (CBMs) as a novel way of measuring the discrepancy in behaviour between processes, taking into account both quantitative aspects and contextual information. This way, process distances by construction take the environment into account: two (non-equivalent) processes may still exhibit very similar behaviour in some contexts, e.g., when certain actions are never performed. We first show how CBMs capture many well-known notions of equivalence and metric, including Larsen's environmental parametrized bisimulation. We then study compositional properties of CBMs with respect to some common process algebraic operators, namely prefixing, restriction, non-deterministic sum, parallel composition and replication.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Process calculi

Keywords and phrases Behavioural metrics, Labelled Transition Systems, Differential Semantics

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.38

Related Version *Full Version*: https://arxiv.org/abs/2307.07400

Funding The first author is supported by the ERC CoG "DIAPASoN", GA 818616, and by PNRR – M4C2 – I1.4 Project CN00000013 "National Centre for HPC, Big Data and Quantum Computing". The second author is supported by MUR project PON REACT EU DM 1062/21.

Acknowledgements We thank the anonymous reviewers for their feedback.

1 Introduction

Simulation and bisimulation relations are often the methodology of choice for reasoning relationally about the behaviour of systems specified in the form of LTSs. On the one hand, most of them can be proved to be congruences, therefore enabling modular equivalence proofs. On the other hand, not being based on any universal quantification (e.g. on tests or on traces), they enable simpler relational arguments, especially when combined with enhancements such as the so-called up-to techniques [29].

The outcome of relational reasoning as supported by (bi)simulation relations is inherently binary: two programs or systems are *either* (bi)similar *or not so*. As an example, all pairs of non-equivalent elements have the same status, i.e. the bisimulation game gives no information on the degree of dissimilarity between non-equivalent states. This can be a problem in those contexts, such as that of probabilistic systems, in which non-equivalent states can give rise to completely different but also extremely similar behaviours.

This led to the introduction of a generalization of bisimulation relations, i.e. the so-called bisimulation *metrics* [7], which rather than being binary relations on the underlying set of states S, are binary *maps* from S to a quantale (most often of real numbers) satisfying the axioms of (pseudo)metrics. In that context, the bisimulation game becomes inherently quantitative: the defender aims at proving that the two states at hand are *close* to each other, while the attacker tries to prove that they are *far apart*. The outcome of this game is a quantity representing a bound not only on any discrepancy about the *immediate* behaviour of the two involved states, (e.g. the fact that some action is available in s but not in t), but

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 38; pp. 38:1–38:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

also providing some information about differences which will only show up in the *future*, all this *regardless* of the actions chosen by the attacker. In this sense, therefore, bisimulation metrics condense a great deal of information in just one number.

Notions of bisimulation metrics have indeed be defined for various sequential and concurrent calculi (see, e.g., [4, 9, 11, 13, 14, 33]), allowing a form of metric reasoning on program behaviour. But when could any of such techniques be said to be compositional? This amounts to be able to *derive* an upper bound on the distance $\delta(C[t], C[s])$ between two programs in the form $C[t]$ and $C[s]$ *from* the distance $\delta(s, t)$ between s and t. Typically, the latter is required to be itself an upper bound on the former, giving rise to *non-expansiveness* as a possible generalization of the notion of a congruence. This, however, significantly *restricts* the class of environments C to which the aforementioned analysis can be applied, since being able to amplify differences is a very natural property of processes. Indeed, an inherent tension exists between expressiveness and compositionality in metric reasoning [14].

But there is another reason why behavioural metrics can be seen as less informative than they could be. As already mentioned, any number measuring the distance between two states s and t implicitly accounts for all the possible ways of comparing s and t, i.e. any context. Often, however, only contexts that act in a certain very specific way could highlight large differences between s and t, while others might simply see s and t as very similar, or even equivalent. This further dimension is abstracted away in compositional metric analysis: if the distance between s and t is very high, but C does not "take advantage" of such large differences, $C[s]$ and $C[t]$ should be *close* to each other, but are dubbed being *far away* from each other, due to the aforementioned abstraction step. It is thus natural to wonder whether metric analysis can be made contextual. In the realm of process equivalences, this is known to be possible through, e.g. Larsen's environmental parametrized bisimulation [23], but not much is known about contextual enhancements of bisimulation *metrics*. Other notions of program equivalence, like logical relations or denotational semantics, have been shown to have metric analogues [6, 30], which in some cases can be made contextual [17, 22].

In this paper, we introduce the novel notion of *contextual behavioural metric* (CBM in the following) through which it is possible to fine-tune the abstraction step mentioned above and which thus represents a refinement over behavioural metrics. In CBMs, the distance between two states s, t of an LTS is measured by an object d having a richer structure than that of a number. Specifically, d is taken to be an element of a metric transition system, in which the contextual and temporal dimensions of the differences can be taken into account. In addition to the mere introduction of this new notion of distance, our contributions are threefold:

- On the one hand, we show that metric labelled transition systems (MLTSs in the following), namely the kind of structures meant to model differences, are indeed quantales, this way allowing us to prove that CBMs are generalized metrics. This is in Section 3.
- On the other hand, we prove that some well-known methodologies for qualitative and quantitative relational reasoning on processes, namely (strong) bisimulation relations and metrics, and environmental parametrized bisimulations [23], can all be seen as CBMs where the underlying MLTS corresponds to the original quantale. This is in Section 4.
- Finally, we prove that CBMs have some interesting compositional properties, and that this allows one to derive approximations to the distance between processes following their syntactic structure. This is in Section 5.

Many of the aforementioned works about behavioural metrics are concerned with probabilistic forms of LTSs. In this work, instead, we have deliberately chosen to focus on usual nondeterministic transition systems. On the one hand, the quantitative aspects can be handled through the so-called *immediate* distance between states, see below. On the other hand, it is well known that probabilistic transition systems can be seen as (non)deterministic systems whose underlying reduction relation is defined between state distributions. Focusing on ordinary LTSs has the advantage of allowing us to concentrate our attention on those aspects related to metrics, allowing for a separation of concerns. This being said, we are confident that most of the results described here could hold for probabilistic LTSs, too.

2 Why the Environment Matters

The purpose of this section is to explain why purely numerical quantales do *not* precisely capture differences between states of an LTS and how a more structured approach to distances can be helpful to tackle this problem. We will do this through an example drawn from the realm of higher-order programs, the latter seen as states of the LTS induced by Abramsky's applicative bisimilarity [1].

Let us start with a pair of programs written in a typed λ-calculus, both of them having type $(\mathtt{Nat} \to \mathtt{Nat}) \to \mathtt{Nat}$, namely M_2 and M_4, where $M_n \triangleq \lambda x.xn$. These terms can indeed be seen as states of an LTS, whose relevant fragment is the following one:

$$\begin{tikzcd}
M_2 \arrow[r, "V"] & V\ 2 \arrow[r, "eval"] & \mathsf{E}(V\ 2) \\
M_4 \arrow[r, "V"] & V\ 4 \arrow[r, "eval"] & \mathsf{E}(V\ 4)
\end{tikzcd}$$

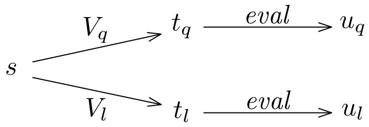
Labelled transitions correspond to either parameter passing (each actual parameter being captured by a distinct label V) or evaluation. It is indeed convenient to see the underlying LTS as a bipartite structure whose states are either computations or values. The two states $\mathsf{E}(V\ 2)$ and $\mathsf{E}(V\ 4)$ are the natural number values to which $V\ 2$ and $V\ 4$ evaluate, respectively. Clearly, the latter are not to be considered equivalent whenever different, and this can be captured, e.g., by either exposing the underlying numerical value through a labelled self-transition or by stipulating that base type values, contrary to higher-order values, can be explicitly observed, thus being equivalent precisely when equal. If one plays the bisimulation game on top of this LTS, the resulting notion of equivalence turns out to be precisely Abramsky's applicative bisimilarity. For very good reasons, M_2 and M_4 are dubbed as *not* equivalent: they can be separated by feeding, e.g. $V = \lambda x.x$ to them.

But now, *how far apart* should M_2 and M_4 be? The answer provided by behavioural metrics consists in saying that M_2 and M_4 are at distance *at most* $x \in \mathbb{R}^{\infty}_{+}$ iff x is an upper bound on the differences any adversary observes while interacting with them, *independently* on how the adversary behaves. As a consequence, if the underlying λ-calculus provides a primitive for multiplication, then it is indeed possible to define values of the form $V_n \triangleq \lambda x.x \times n$ for every n, allowing the environment to observe arbitrarily large differences of the form

$$|\ \mathsf{E}(V_n\ 2) - \mathsf{E}(V_n\ 4)\ | = |\ 2n - 4n\ | = 2n.$$

In other words, the distance between M_2 and M_4 is $+\infty$. The possibility of arbitrarily amplifying distances is well-known, and can be tackled, e.g., by switching to a calculus in which *all* functions are *non-expansive*, ruling out terms such as V_n where $n > 1$. In other words, the distance between M_2 and M_4 is indeed 2, because no input term V can "stretch" the distance between 2 to 4 to anything more than 2. This is what happens, e.g., in Fuzz [30].

But is this the end of the story? Are we somehow losing too much information by stipulating that M_2 and M_4 are, say, at distance 2? Actually, the only moment in which the environment observes the state with which it is interacting is at the end of the dialogue, namely after feeding it with a function $V : \texttt{Nat} \to \texttt{Nat}$. If, for example, the environment picks $V_q \triangleq \lambda x.(x-3)^2+2$, then the observed difference is 0, while if it picks $V_l \triangleq \lambda x.x+2$ then the observed distance is maximal, i.e. 2. In other words, the observed distance strictly depends on how the environment behaves and should arguably be parametrised on it. This is indeed the main idea behind Larsen's environmental parametrised bisimulation, but also behind our contextual behavioural metrics. In the latter, differences can be faithfully captured by the states of *another* labelled transition system, called a metric labelled transition system, in which observed distances are associated to states. In our example, the difference between M_2 and M_4 *is* the state s of a metric labeled transition system whose relevant fragment is:



Crucially, while s, t_s, t_l, u_l are all mapped to the null observable difference, u_q is associated to 2. This allows to discriminate between those environments which are able to see large differences from those which are not. This is achieved by allowing differences to be modelled by the states of a transition system *themselves*. Using a categorical jargon, it looks potentially useful, but also very tempting, to impose the structure of a coalgebra to the underlying space of distances rather than taking it as a monolithical, numeric, quantale. The rest of this paper can be seen as an attempt to make this idea formal.

3 Contextual Behavioural Metrics, Formally

This section is devoted to introducing contextual behavioural metrics, namely the concept we aim at studying in this paper. We start with the definition of quantale [31], the canonical codomain of generalized metrics [24]. The notion of quantale used in this paper is that of unital integral commutative quantale:

▶ **Definition 1** (Quantale). *A* quantale *is a structure* $\mathbb{Q} = (Q, \bigwedge, \bigvee, \bot, \top, +)$ *such that* $\bigwedge, \bigvee : 2^Q \to Q$*, the two objects* $\bot, \top$ *are in* Q*, and* $+$ *is a binary operation on* Q*, where:*

- $(Q, \bigwedge, \bigvee, \bot, \top)$ *is a complete lattice;*
- $(Q, +, \bot)$ *is a commutative monoid;*
- *for every* $e \in Q$ *and every* $A \subseteq Q$ *it holds that* $e + \bigwedge A = \bigwedge\{e + f \mid f \in A\}$.

We write $e \leq f$ *when* $e = \bigwedge\{e, f\}$.

Generalized metrics are maps which associate an element of a given quantale to each pair of elements. As customary in behavioural metrics, we work with *pseudo*metrics, in which distinct elements may be at minimal distance:

▶ **Definition 2** (Metrics). *A* pseudometric *over a set* A *with values in a quantale* $\mathbb{Q}$ *is a map* $m : A \times A \to \mathbb{Q}$ *satisfying:*

- *for all* $a \in A : m(a,a) = \bot$*;*
- *for all* $a, b \in A : m(a,b) = m(b,a)$*;*
- *for all* $a, b, c \in A : m(a,c) \leq m(a,b) + m(b,c)$*.*

In the rest of this paper, we refer to pseudometrics simply as metrics.

It is now time to introduce our notion of a *process*, namely of the computational objects we want to compare. We do not fix a syntax, and work with abstract labelled transition systems (LTSs in the following). In order to enable (possibly quantitative) metric reasoning, we equip states of our LTS with an immediate metric D, namely a metric measuring the observable distance between two states.

▶ **Definition 3** (Process LTS). *We define a $\mathbb{Q}$-LTS as a quadruple $(P, \mathcal{L}, \rightarrow, D)$ where:*

- *P is the set of processes;*
- *$\mathcal{L}$ is the set of labels;*
- *$\rightarrow \subseteq P \times \mathcal{L} \times P$ is the transition relation;*
- *$D : P \times P \rightarrow \mathbb{Q}$ is a metric.*

▶ **Example 4.** The example LTS from Section 2 should be helpful in understanding why the metric D is needed: terms and values of distinct types are at maximal immediate distance, while terms and values of the same type are at minimal distance, except when the type is `Nat`, whereas the immediate distance is just the absolute value between the two numbers.

We now need to introduce *another* notion of transition system, this time meant to model *differences* between computations. This kind of structure can be interpreted as a quantale, and will form the codomain of Contextual Bisimulation Metrics. Intuitively, a Metric LTS is an LTS endowed with a function from states to a quantale $\mathbb{Q}$. This allows to keep track of immediate distance changes. Let us start with the notion of a *pre*-metric LTS:

▶ **Definition 5** (Pre-metric LTS). *A* pre-metric $\mathbb{Q}$-LTS *is a quadruple $\mathbb{V} = (S, \mathcal{L}, \rightarrow, \Downarrow)$ where:*

- *S is the set of states;*
- *$\mathcal{L}$ is the set of labels;*
- *$\rightarrow \subseteq S \times \mathcal{L} \times S$ is the transition relation;*
- *$\Downarrow : S \rightarrow \mathbb{Q}$ is a function which assigns values in $\mathbb{Q}$ to states in S.*

A pre-metric LTS does not necessarily form a quantale, because S does not necessarily have, e.g. the structure of a monoid or a lattice. In order to be proper codomains for metrics, pre-metric LTSs need to be endowed with some additional structure, which will be proved to be enough to form a quantale.

▶ **Definition 6** (Metric LTS). *A metric $\mathbb{Q}$-LTS $\mathbb{V} = (S, \mathcal{L}, \rightarrow, \Downarrow)$ is a pre-metric $\mathbb{Q}$-LTS endowed with two elements $\bot_{\mathbb{V}}, \top_{\mathbb{V}} \in S$, and three operators $\bigwedge_{\mathbb{V}}, \bigvee_{\mathbb{V}} : 2^S \rightarrow S$ and $+_{\mathbb{V}} : S \times S \rightarrow S$, where the conditions hold for all possible values of the involved metavariables:*

$$\begin{array}{cc}
\bot_{\mathbb{V}} \xrightarrow{\ell} s \iff s = \bot_{\mathbb{V}} & \Downarrow \bot_{\mathbb{V}} = \bot_{\mathbb{Q}} \\
\forall \ell \in \mathcal{L} : \top_{\mathbb{V}} \overset{\ell}{\not\rightarrow} & \Downarrow \top_{\mathbb{V}} = \top_{\mathbb{Q}} \\
\bigwedge_{\mathbb{V}} S' \xrightarrow{\ell} s \iff \exists s' \in S' : s' \xrightarrow{\ell} s & \Downarrow \bigwedge_{\mathbb{V}} S' = \bigwedge_{\mathbb{Q}} \{\Downarrow s \mid s \in S'\} \\
\bigvee_{\mathbb{V}} S' \xrightarrow{\ell} s \iff \exists S'' : s = \bigvee_{\mathbb{V}} S'' \textit{ and} & \Downarrow \bigvee_{\mathbb{V}} S' = \bigvee_{\mathbb{Q}} \{\Downarrow s \mid s \in S'\} \\
\exists \textit{ surjective } f : S' \rightarrow S'' : \forall s' \in S' : s' \xrightarrow{\ell} f(s') & \\
s_1 +_{\mathbb{V}} s_2 \xrightarrow{\ell} s' \iff s' = s'_1 +_{\mathbb{V}} s'_2 \textit{ for some } s'_1, s'_2 & \Downarrow (s_1 +_{\mathbb{V}} s_2) = \Downarrow s_1 +_{\mathbb{Q}} \Downarrow s_2 \\
\textit{such that: } s_1 \xrightarrow{\ell} s'_1 \textit{ and } s_2 \xrightarrow{\ell} s'_2 &
\end{array}$$

Axioms ensures that $\bot_{\mathbb{V}}$ allows every possible behaviour (somehow capturing every context), and dually $\top_{\mathbb{V}}$ disallows every behaviour. $\bigwedge_{\mathbb{V}} S'$ allows all and only the behaviours in S' (union of contexts), while $\bigvee_{\mathbb{V}} S'$ enables all and only the behaviours allowed by *every* element in S' (intersection of contexts). The sum $+_{\mathbb{V}}$ has a behaviour similar to $\bigvee_{\mathbb{V}}$, but it is binary and differs on the value returned by $\Downarrow$.

▶ Remark 7 (On The Existance Of Non-Trivial MLTSs). Due to the requirements about joins and meets over potentially infinite sets, MLTSs are not easy to define directly. We argue, however, that an MLTS can be defined as the closure of a pre-MLTS. If the underlying quantale $\mathbb{Q}$ is boolean, one can get the desired structure by considering 2^{2^X}, where X is the carrier of the given pre-MLTS: it suffices to take subsets in "conjunctive" normal form. For the general case, the class $\cup_{n\in\mathbb{N}} \overbrace{2^{\cdot^{\cdot^{\cdot^{2^X}}}}}^{n \text{ times}}$, which is indeed a set in ZFC, suffices.

The axiomatics above is still not sufficient to give the status of a quantale to $\mathbb{Q}$-MLTSs. The reason behind all this is that there could be equivalent but distinct states in S. We then define a preorder $\leq_{\mathbb{V}}$ on the states of any MLTS $\mathbb{V}$:

▶ **Definition 8.** *A relation $\mathcal{R} \subseteq S \times S$ is a* $\leq_{\mathbb{Q}}$-preserving simulation[1] *if, whenever $s_1 \; \mathcal{R} \; s_2$, it holds that:*

1. $\Downarrow s_1 \; \leq_{\mathbb{Q}} \Downarrow s_2$*;*
2. $\forall \ell \in \mathcal{L} : s_2 \xrightarrow{\ell} s_2' \implies \exists s_1' : s_1 \xrightarrow{\ell} s_1'$ *and* $s_1' \mathcal{R} s_2'$*.*

We define $\leq_{\mathbb{V}} \subseteq S \times S$ as the largest $\leq_{\mathbb{Q}}$-preserving simulation. We use the notation $\leq\geq_{\mathbb{V}}$ for mutual $\leq_{\mathbb{Q}}$-preserving simulation, that is $\leq\geq_{\mathbb{V}} = \leq_{\mathbb{V}} \cap \geq_{\mathbb{V}}$. We say that s is a lower (resp. upper) bound of $S' \subseteq S$ if $s \leq_{\mathbb{V}} s'$ (resp. $s' \leq_{\mathbb{V}} s$) for all $s' \in S'$.

The forthcoming result states that, in general, MLTSs *almost* form quantales. We can recover a proper quantale by quotienting S modulo $\leq\geq_{\mathbb{V}}$.

▶ **Proposition 9** (Properties of MLTSs). *Let $\mathbb{V} = (S, \mathcal{L}, \rightarrow, \Downarrow)$ be a MLTS. Then:*

1. $\leq_{\mathbb{V}}$ *is a preorder relation;*
2. *For all s: $\bot_{\mathbb{V}} \leq_{\mathbb{V}} s$ and $s \leq_{\mathbb{V}} \top_{\mathbb{V}}$;*
3. *For all $S' \subseteq S$: $\bigwedge_{\mathbb{V}} S'$ is a lower bound of S', and if s' is a lower bound of S' then $s' \leq_{\mathbb{V}} \bigwedge_{\mathbb{V}} S'$.*
4. *For all $S' \subseteq S$: $\bigvee_{\mathbb{V}} S'$ is an upper bound of S', and if s' is an upper bound of S' then $\bigwedge_{\mathbb{V}} S' \leq_{\mathbb{V}} s'$.*
5. *For all $s \in S, S' \subseteq S : s +_{\mathbb{V}} \bigwedge_{\mathbb{V}} S' \leq\geq_{\mathbb{V}} \bigwedge_{\mathbb{V}} \{s + s' \mid s' \in S'\}$.*
6. *For all $s \in S : s +_{\mathbb{V}} \bot \leq\geq_{\mathbb{V}} s$.*
7. *For all $s, s' \in S : s +_{\mathbb{V}} s' \leq\geq_{\mathbb{V}} s' + s$.*
8. *For all $s, s', s'' \in S : (s +_{\mathbb{V}} s') +_{\mathbb{V}} s'' \leq\geq_{\mathbb{V}} s' +_{\mathbb{V}} (s +_{\mathbb{V}} s'')$.*
9. *If $\leq_{\mathbb{V}}$ is a partial order relation, then $\mathbb{V}$ is a quantale.*

Unless stated otherwise, we assume that every MLTS $\mathbb{V}$ we work with is a quantale.

▶ **Definition 10** (Contextual Behavioural Metrics). *Let $(P, \mathcal{L}, \rightarrow, D)$ and $\mathbb{V} = (S, \mathcal{L}, \rightarrow, \Downarrow)$ be, respectively, a $\mathbb{Q}$-LTS and a $\mathbb{Q}$-MLTS. Then, a map $m : P \times P \rightarrow S$ is a* contextual bisimulation map *if:*

1. $D(p, q) \; \leq_{\mathbb{Q}} \; \Downarrow m(p, q)$*;*
2. *if $m(p, q) \xrightarrow{\ell} s'$, then the following holds:*
   - **a.** $p \xrightarrow{\ell} p' \implies \exists q' : q \xrightarrow{\ell} q'$ *and* $m(p', q') \leq_{\mathbb{V}} s'$*;*
   - **b.** $q \xrightarrow{\ell} q' \implies \exists p' : p \xrightarrow{\ell} p'$ *and* $m(p', q') \leq_{\mathbb{V}} s'$*.*

We say that m is a contextual bisimulation metric *(CBM) if m is both a contextual bisimulation map and a metric. We define the contextual bisimilarity map δ as follows:*

$$\delta(p, q) = \bigwedge_{\mathbb{V}} \{m(p, q) \mid m \text{ is a contextual bisimulation map}\}$$

[1] Technically, it is a reverse simulation. We call it simulation for brevity.

The following result states that the contextual bisimilarity map is well behaved, being a contextual bisimulation map upper bounding any other such map:

▶ **Lemma 11.** *δ is a contextual bisimulation map. Moreover, for all contextual bisimulation maps m, and processes p, q, it holds that $\delta(p, q) \leq_{\mathbb{V}} m(p, q)$.*

We still do not know whether δ is a *metric*. We need a handy characterization of δ for that.

A Useful Characterization of CBMs

Larsen's environment parametrized bisimulations [23] is a variation on ordinary bisimulation in which the compared states are tested against environments of a specific kind, this way giving rise to a ternary relation. We here show that CBMs can be captured along the same lines. A formal comparison between CBMs and Larsen's approach is deferred to Section 4.3.

▶ **Definition 12** (Parametrized Bisimulation). *Let $(P, \mathcal{L}, \rightarrow, D)$ and $(S, \mathcal{L}, \rightarrow, \Downarrow)$ be, respectively, a $\mathbb{Q}$-LTS and a $\mathbb{Q}$-MLTS. An S-indexed family of relations $\{\mathcal{R}_s\}$ such that $\mathcal{R}_s \subseteq P \times P$ is said to be a* parametrized bisimulation *iff, whenever $p \; \mathcal{R}_s \; q$, it holds that $D(p, q) \leq_{\mathbb{Q}} \Downarrow s$, and $s \xrightarrow{\ell} s'$ implies:*

- *$p \xrightarrow{\ell} p' \implies \exists q' : q \xrightarrow{\ell} q'$ and $p' \mathcal{R}_{s'} q'$;*
- *$q \xrightarrow{\ell} q' \implies \exists p' : p \xrightarrow{\ell} p'$ and $p' \mathcal{R}_{s'} q'$.*

Parametrized bisimilarity is the largest parametrized bisimulation, namely the largest family $\{\sim_s\}$ such that $p \sim_s q$ if $p \mathcal{R}_s q$ for some parametrized bisimulation $\{\mathcal{R}_s\}$.

The fact that $\{\sim_s\}$ is indeed a parametrized bisimulation holds because parametrized bisimulations are closed under unions (defined point-wise), something which can be proved with a simple generalisation of standard techniques [26, 27]. Parametrized bisimilarity turns out to be strongly related to δ, this way providing a simple proof technique that will be heavily used in the rest of the paper.

▶ **Proposition 13.** *For all p, q, s, it holds that $\delta(p, q) \leq_{\mathbb{V}} s \iff p \sim_s q$.*

We are finally ready to state that δ satisfies the axioms of a metric.

▶ **Theorem 14.** *The contextual bisimulation map δ is a metric.*

4 Some Relevant Examples

This section is devoted to showing how well-known and heterogeneous notions of equivalence and distance can be recovered as CBMs for appropriate quantales and MLTSs.

4.1 Strong Bisimilarity as a CBM

We start recalling that strong bisimilarity [26, 27] is the largest strong bisimulation relation, that is a relation $\mathcal{R} \subseteq P \times P$ on the states of a plain LTS $(P, \mathcal{L}, \rightarrow)$ such that $p \; \mathcal{R} \; q$ implies:

- $p \xrightarrow{\ell} p' \implies \exists q' : q \xrightarrow{\ell} q'$ and $p' \; \mathcal{R} \; q'$;
- $q \xrightarrow{\ell} q' \implies \exists p' : p \xrightarrow{\ell} p'$ and $p' \; \mathcal{R} \; q'$.

The first thing we have to do to turn strong bisimilarity into a CBM is to define, given such an LTS $(P, \mathcal{L}, \rightarrow)$, a canonical immediate distance D on the boolean quantale $\mathbb{B}$, which we call the *canonical* distance:

$$D(p, q) = \begin{cases} \bot \text{ if } \forall \ell : p \xrightarrow{\ell} \iff q \xrightarrow{\ell} \\ \top \text{ otherwise} \end{cases}$$

That is, the immediate distance is $\perp$ precisely when the processes expose the same labels. Notice that immediate distance is not affected by possible future behavioural differences. Any LTS like this is said to be a *boolean* LTS. The boolean quantale can be turned very naturally into a MLTS: let $\mathbb{V}$ be $(\{\perp_{\mathbb{V}}, \top_{\mathbb{V}}\}, \mathcal{L}, \to, \Downarrow)$ where the transitions are self loops $\perp_{\mathbb{V}} \xrightarrow{\ell} \perp_{\mathbb{V}}$ for every $\ell \in \mathcal{L}$, and $\Downarrow$ just associates $\perp_{\mathbb{Q}}$ to $\perp_{\mathbb{V}}$ and $\top_{\mathbb{Q}}$ to $\top_{\mathbb{V}}$.

▶ **Proposition 15.** *Given any boolean LTS, δ is the characteristic function of bisimilarity, i.e. $\delta(p,q) = \perp_{\mathbb{V}} \iff p \sim q$.*

4.2 Behavioural CBMs

Most behavioural metrics from the literature are defined on *probabilistic* transition systems [8, 10, 34], differently from CBMs. Some probabilistic behavioural metrics can still be captured in our framework by using as states of the process LTS (sub)distributions of states of the original PLTS, e.g. the distribution based metric in [12]. Non-probabilistic behavioural metrics exist, e.g., the so-called "branching metrics" [5], which are indeed instances of behavioural metrics as defined below. Notice that our definition has a generic quantale $\mathbb{Q}$ as its codomain, while usually behavioural metrics take values in the interval $\mathbb{R}_{[0,1]}$.

Let us first recall what we mean by a behavioural metric here. A metric $M : P \times P \to \mathbb{Q}$ is said to be a *behavioural metric* if, for all pairs of states p, q, it holds that $D(p,q) \leq_{\mathbb{Q}} M(p,q)$ and, whenever $M(p,q) <_{\mathbb{Q}} \top_{\mathbb{Q}}$, we have that:

- $p \xrightarrow{\ell} p' \implies \exists q' : q \xrightarrow{\ell} q'$ and $M(p,q) \geq_{\mathbb{Q}} M(p',q')$;
- $q \xrightarrow{\ell} q' \implies \exists p' : p \xrightarrow{\ell} p'$ and $M(p,q) \geq_{\mathbb{Q}} M(p',q')$.

Intuitively, behavioural metrics can be seen as quantitative variations on the theme of a bisimulation: they associate a value from a quantale to each pair of processes (rather than a boolean), they are coinductive in nature. Moreover, they are based on the bisimulation game, i.e., any move of one of the two processes needs to be matched by some move of the other, at least when their distance is not maximal. Our definition is similar to the one in [12]. However, many behavioural metrics in literature deal with non-determinism through the Hausdorff lifting, that is by stipulating that $D(p,q) \leq_{\mathbb{Q}} M(p,q)$ and for all ℓ:

$$M(p,q) \geq_{\mathbb{Q}} \bigvee_{p \xrightarrow{\ell} p'} \bigwedge_{q \xrightarrow{\ell} q'} M(p',q') \qquad \text{and} \qquad M(p,q) \geq_{\mathbb{Q}} \bigvee_{q \xrightarrow{\ell} q'} \bigwedge_{p \xrightarrow{\ell} p'} M(p',q')$$

The two notions are equivalent if the process LTS is image-finite and $\mathbb{Q}$ is totally ordered, both conditions are often assumed to be true in the literature.

We now show how to interpret $\mathbb{Q}$ as a MLTS. Morally, we just fix $\mathbb{Q}$ as the set of states, the identity as $\Downarrow$, and self loops as transitions. This however violates the requirement that the top element has no outgoing transitions. We therefore add the element $\top_{\mathbb{V}}$. Notice that we still need $\top_{\mathbb{Q}}$, as it ensures that $\mathbb{V}$ is closed under $+_{\mathbb{V}}$. Let $\mathbb{V} = (S, \mathcal{L}, \to, \Downarrow)$ where $S = \mathbb{Q} \uplus \{\top_{\mathbb{V}}\}$, $\mathcal{L}$ is as in the underlying process LTS, transitions are the self loops of the form $s \xrightarrow{\ell} s$ for every $\ell \in \mathcal{L}$, and $s \in \mathbb{Q}$, $\Downarrow$ is the identity on $\mathbb{Q}$, and $\Downarrow(\top_{\mathbb{V}}) = \top_{\mathbb{Q}}$. Notice that, when $\Downarrow s <_{\mathbb{Q}} \top_{\mathbb{Q}}$, we have that:

$$\Downarrow s \leq_{\mathbb{Q}} \Downarrow s' \iff s \leq_{\mathbb{V}} s'. \tag{1}$$

We also have that for every behavioural metric there is a CBM that "agrees" on the quantitative distance between processes. This intuition is formalized as follows:

▶ **Proposition 16.** *Let M be a behavioural metric, and let m_M be defined as:*

$$m_M(p,q) = \begin{cases} M(p,q) & \textit{if } M(p,q) <_{\mathbb{Q}} \top_{\mathbb{Q}} \\ \top_{\mathbb{V}} & \textit{otherwise} \end{cases}$$

Then, m_M is a CBM and for every p, q it holds that $\Downarrow m_M(p,q) = M(p,q)$.

The agreement of M and m_M holds by definition. The fact that m_M is a CBM, instead is a consequence of the fact that transitions preserve M distances (by definition of behavioural metric), and that behavioural metrics are metrics, indeed.

4.3 On Environment Parametrised Bisimulation and CBMs

As already mentioned, the concept of a CBM is inspired by Larsen's environment parametrized bisimulation [23]. It should then come with no surprise that there is a relationship between the two, which is the topic of this section.

First, let us recall what an environment parametrized bisimulation is. Let $(P, \mathcal{L}, \rightarrow)$ and $(E, \mathcal{L}, \rightarrow)$ be LTSs. Elements of P are called processes, while elements of E are called environments. A E-indexed family of relations $\{\mathcal{R}_e\}$, where $\mathcal{R}_e \subseteq P \times P$ is a *environment parametrized bisimulation* (EPB in the following) if, whenever $p\ \mathcal{R}_e\ q$ and $e \xrightarrow{\ell} e'$:

- $p \xrightarrow{\ell} p' \implies \exists q' : q \xrightarrow{\ell} q'$ and $p' \mathcal{R}_{e'} q'$;
- $q \xrightarrow{\ell} q' \implies \exists p' : p \xrightarrow{\ell} p'$ and $p' \mathcal{R}_{e'} q'$.

Environment parametrized bisimilarity, denoted as $\sim_e$, is defined as $p \sim_e q$ iff $p\ \mathcal{R}_e\ q$ for some EPB $\mathcal{R}$. It turns out that $\sim_e$ is the largest EPB [23].

EPBs can be embedded into the CBMs framework as follows:

- fix $\mathbb{Q}$ as the boolean quantale $\mathbb{B}$, and define $D(p,q) = \begin{cases} \bot_{\mathbb{B}} & \text{if } \exists \ell : p \xrightarrow{\ell} \text{ and } q \xrightarrow{\ell} \\ \top_{\mathbb{B}} & \text{otherwise} \end{cases}$
- let $\mathbb{V}_E = (S, \mathcal{L}, \rightarrow, \Downarrow)$ be any MLTS such that for all $s \in S$ it holds that $\Downarrow s = \bot_{\mathbb{B}} \iff s \neq \top_{\mathbb{V}}$, and for all $e \in E$ there is $s_e \in S$ such that $e \precsim\succsim s_e$. Here $\precsim\succsim$ is strong mutual similarity on the disjoint union of $\mathbb{V}$ (forgetting $\Downarrow$) and E. When such conditions hold, we say that E is embedded into $\mathbb{V}_E$.

We remark that, for every E, there is an MLTS $\mathbb{V}_E$ enjoying the properties above, obtained by augmenting E with the immediate metric defined above (this gives rise to a pre-metric LTS, Definition 5) and by closing it with respect to the operations and constants $\bigvee, \bigwedge, \top, \bot$ of Definition 6. The intuition is that:

- Two processes should have minimal immediate distance if there is a non-empty context in which their immediate behaviour is equivalent. This is ensured by the fact that they exhibit at least a common label from their current state.
- $\mathbb{V}_E$ needs to precisely simulate the behaviours in E. We therefore require that every element of E has a corresponding element in s, with "equivalent behaviour". In this setting, mutual simulation turns out to be the appropriate notion of behavioural equivalence.

The link between environment parametrized bisimulations and CBMs is made formal by the following proposition.

▶ **Proposition 17.** *Let E be an environment LTS embedded into an MLTS $\mathbb{V}_E$. For every p, q and e, it holds that $p \sim_e q \implies \delta(p,q) \leq_{\mathbb{V}_E} s_e$.*

The proposition above ultimately follows from the fact that $p \sim_e q \iff p \sim_{s_e} q$ (where $\sim_{s_e}$ is parametrized bisimilarity Definition 12) together with Proposition 13.

5 About the Compositionality of CBMs

One of the greatest advantages of the bisimulation proof method is its modularity, which comes from the fact that, under reasonable assumptions, bisimilarity is a congruence. In a metric setting, one strives to obtain similar properties [14, 16], which take the form of non-expansiveness, or variations thereof.

In this section we study the compositionality properties of CBMs with respect to some standard process algebraic operators. We are interested in properties that generalise the concept of a congruence. Following the lines of [22, 28], our treatment will be contextual, meaning that the environment in which processes are deployed can indeed contribute to altering their distance, although in a controlled way.

In order to keep our theory syntax independent, we model operators f as functions $f : P^n \to P$ (where n is the arity of the operator). In particular, for each process operator f of arity n we define the function $\hat{f} : P^n \times S^n \to S$ as follows:

$$\hat{f}(p_1, \dots, p_n, s_1, \dots, s_n) = \bigvee_{\mathbb{V}} \{\delta(f(p_1, \dots, p_n), f(p'_1, \dots, p'_n)) \mid \forall 1 \leq i \leq n : \delta(p_i, p'_i) \leq_{\mathbb{V}} s_i\}$$

Intuitively, $\hat{f}(\vec{p}, \vec{s})$ bounds $\delta(f(\vec{p}), f(\vec{q}))$ whenever $\vec{q}$ is such that $\delta(p_i, q_i) \leq_{\mathbb{V}} s_i$ for every i. Moreover, $\hat{f}(\vec{p}, \vec{s})$ is the lowest among such bounds.

Of course, our compositionality results rely on some assumptions on the compositionality of the immediate metric D. Formally, we require that, for all operators f (with arity n), the following holds for every $p_1, \dots, p_n, q_1, \dots, q_n$:

$$D(f(p_1, \dots, p_n), f(q_1, \dots, q_n)) \leq_{\mathbb{Q}} D(p_1, q_1) +_{\mathbb{Q}} \dots +_{\mathbb{Q}} D(p_n, q_n). \tag{2}$$

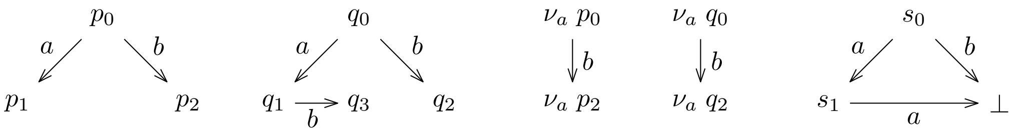
Below, we will give results about when and under which condition the value of the operator $\hat{f}$ can be upper-bounded by a function on its parameters. We remark that our compositionality results apply to each operator independently.

For the sake of concreteness, we give some examples of processes and their metric analysis. To this purpose, let $\mathcal{L} = \{a, b\}$, fix $\mathbb{Q}$ as the boolean quantale $\mathbb{B}$ and let D be defined exactly as we did in Section 4.3 (i.e., D returns $\bot$ if the processes can fire some common action, $\top$ otherwise). Distances will take values from a MLTS $\mathbb{V}_0$ over $\mathbb{B}$. Similarly to Section 4.3, we require $\mathbb{V}_0$ to be such that for every $s \in S$ it holds that $\Downarrow s = \bot_{\mathbb{B}} \iff s \neq \top_{\mathbb{V}_0}$. Moreover, we assume that $\mathbb{V}_0$ is able to represent at least Milner's synchronisation trees [25]. For simplicity, we omit self loops of $\bot_{\mathbb{V}_0}$ from all the graphical representations of our MLTS. Of course, these assumptions hold only in the examples, while our results hold for general MLTSs.

5.1 Restriction

We assume restriction to be modelled by a $\mathcal{L}$-indexed family of unary operators ν_ℓ, and that P is closed under these operators. Their semantics is standard.

▶ **Example 18.** Let p_0 and q_0 be as in the following figure. We have that p_0 and q_0 have the exact same behaviour on the b branch, while we can observe differences on the a branch (q_1 can perform an action, p_1 is terminated). State s_0 captures exactly the similarities between p_0 and q_0: after a b move it reduces to $\bot$; after an a move, it reduces to s_1. We argue that s_1 captures the similarities between p_1 and q_1: since neither of the two can perform the action a, s_1 reduces to $\bot$ with label a, while it does not perform b actions because p_1 and q_1 "disagree" on such label. So $\delta(p_0, q_0) = s_0$. Processes $(\nu a)p_0$ and $(\nu a)q_0$ exhibit equivalent behaviour instead. In fact, operator (νa) filters out the problematic a branch. It is therefore the case that $\delta((\nu a)p_0, (\nu a)q_0) = \bot$.



⌟

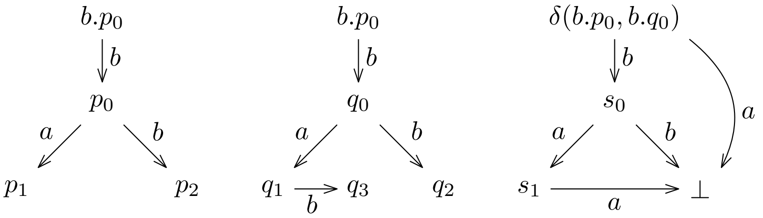
The restriction operator does not add new behaviours to the original process, as it can only restrict it. We can then expect that the differences between any two processes do not increase if such processes are placed in a restriction context. Proposition below indeed shows that $\hat{\nu_\ell}$ enjoys a property similar to non-expansiveness, that is the distance between any two processes p and q bounds the distance between $\nu_\ell\ p$ and $\nu_\ell\ q$.

▶ **Proposition 19.** $\hat{\nu_\ell}(p, s) \leq_{\mathbb{V}} s$.

5.2 Prefixing

We assume that P is closed under operator $. : \mathcal{L} \times P \to P$, whose semantics is standard. We proceed similarly to the case of ν: we treat the prefix operator . as an $\mathcal{L}$-indexed family of unary operators $._\ell$.

▶ **Example 20.** Let p_0 and q_0 be as in Example 18. Since $b.p_0$ and $b.q_0$ can only reduce with a b move to, respectively, p_0 and q_0, their distance $\delta(b.p_0, b.q_0)$ should reduce to $\delta(p_0, q_0) = s_0$. Moreover, after performing an a action, $\delta(b.p_0, b.q_0)$ should reduce to $\bot$.



⌟

In our contextual setting, prefixing of processes can change the distance, and the new distance may be incomparable to the original one. Therefore properties like non-expansiveness do not hold in general for $\hat{.}_\ell$. Among the compositionality properties appeared in literature, uniform continuity [15] seems appropriate for prefixing. Uniform continuity holds when for all $s_\epsilon >_{\mathbb{V}} \bot_{\mathbb{V}}$ there is $s_\delta >_{\mathbb{V}} \bot_{\mathbb{V}}$ such that $\hat{.}_\ell(p, s_\delta) \leq_{\mathbb{V}} s_\epsilon$. Such condition is too strong: for instance if $s_\epsilon \xrightarrow{\ell} \bot_{\mathbb{V}}$ the only option is to take $s_\delta = \bot_{\mathbb{V}}$, hence $s_\delta \not>_{\mathbb{V}} \bot_{\mathbb{V}}$.

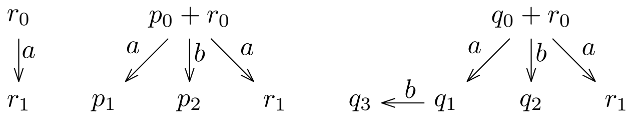
For this reason, we need a stronger property for s_ϵ, namely that the meet of the set of ℓ reducts of s_ϵ is strictly greater than $\bot_{\mathbb{V}}$ and its immediate value is lower than that of s_ϵ.

▶ **Proposition 21.** *For all $s_\epsilon >_{\mathbb{V}} \bot_{\mathbb{V}}$ such that $s_\ell = \bigwedge_{\mathbb{V}}\{s \mid s_\epsilon \xrightarrow{\ell} s\} > \bot$ and $\Downarrow s_\ell \leq_{\mathbb{Q}} \Downarrow s_\epsilon$, there is $s_\delta >_{\mathbb{V}} \bot_{\mathbb{V}}$ such that $\hat{.}_\ell(p, s_\delta) \leq_{\mathbb{V}} s_\epsilon$.*

5.3 Non-deterministic Sum

We assume that P is closed under binary operator $+$, whose semantics is again standard.

▶ **Example 22.** Let p_0, q_0 and s_0 be as in Example 18, and r_0 as in the picture below. We have that $\delta(p_0 + r_0, q_0 + r_0) = s_0$: it reduces to $\bot$ after a b move (both processes indeed terminate after a b action). An a action instead leads to a state that can only perform a a action towards $\bot$. This is because $q_0 + r_0$ can reduce to q_1 with a a move, while $p_0 + r_0$ cannot match that action exactly: it can reduce to p_1 or r_1, that are not bisimilar to q_1.



┘

Intuitively, the non-deterministic sum of two precesses can behave as the former process or as the latter (but not as both). Therefore we can expect that the distance between two sums is bounded by the join of the distances of the components. This is however not always the case, as the immediate distance is not necessarily non-expansive. The sum operator $+_{\mathbb{V}}$ from Definition 6, instead, turns out to be sufficient for our purposes. Proposition below indeed shows that $\hat{+}$ is non-extensive.

▶ **Proposition 23.** *For every p_1, p_2, s_1, s_2 : it holds that $\hat{+}(p_1, p_2, s_1, s_2) \leq_{\mathbb{V}} s_1 +_{\mathbb{V}} s_2$.*

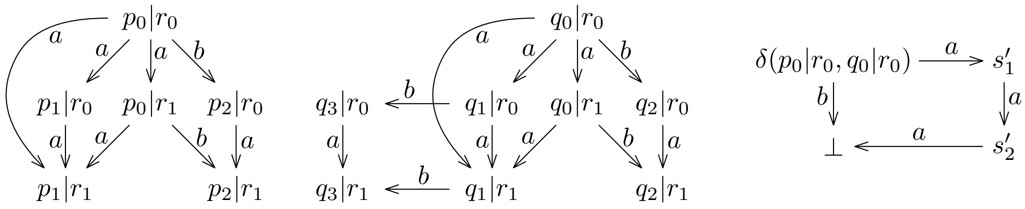
5.4 Parallel Composition

We assume P to be closed under the binary operator $|$, whose semantics is defined below:

$$\frac{p \xrightarrow{\ell} p'}{p|q \xrightarrow{\ell} p'|q} \qquad \frac{p \xrightarrow{\ell} p' \quad q \xrightarrow{\ell} q'}{p|q \xrightarrow{\ell} p'|q'} \qquad \frac{q \xrightarrow{\ell} q'}{p|q \xrightarrow{\ell} p|q'}$$

The notion of synchronisation considered in this paper is the one pioneered in CSP [19, 35]. This choice is motivated by the fact that, in comparison with CCS-like communication [25] (which requires dual actions to synchronise resulting in an invisible τ-action), CSP notion does not change the label: this simplifies the technical development and enables stronger compositionality properties. Most of the works on compositionality of metrics for parallel composition we are aware of use CSP synchronisation, e.g. [2, 14, 15].

▶ **Example 24.** Let p_0 and q_0 be as in Example 18, and r_0 as in Example 22. We have that $\delta(p_0|r_0, q_0|r_0)$ is as the figure below. Indeed, $p_0|r_0$ and $q_0|r_0$ necessarily reduce to bisimilar states after a b action: therefore their distance b-reduces to $\bot$. The situation for a actions is more involved, due the the presence of several a-reducts for both processes. So, consider the transition $p_0|r_0 \xrightarrow{a} p_1|r_0$. We need to find the matching move of $q_0|r_0$ that minimises the distance between the reducts. So, consider the transition $q_0|r_0 \xrightarrow{a} q_1|r_1$. Since $p_1|r_0$ can only perform a actions while $q_1|r_1$ only b ones, we have that $\delta(p_1|r_0, q_1|r_1) = \top$. If we instead consider transition $q_0|r_0 \xrightarrow{a} q_1|r_0$, we have that $\delta(p_1|r_0, q_1|r_0) = s'_1$. Indeed, $q_1|r_0 \xrightarrow{b}$ while $p_1|r_0$ does not: hence $s'_1 \not\xrightarrow{b}$. Moreover, $s'_1 \xrightarrow{a} s'_2$. The only a-reducts of $p_1|r_0$ and $q_1|r_0$ are, respectively, $p_1|r_1$ and $q_1|r_1$. It is easy to verify that $\delta(p_1|r_1, q_1|r_1) = s'_2$. The last possible matching choice is $q_0|r_0 \xrightarrow{a} q_0|r_1$, for which we have that $\delta(p_1|r_0, q_0|r_1) = s'_1$: the argument is similar to the previous case. All the other starting a-moves of $p_0|r_0$, and those of $q_0|r_0$, have matching moves leading to distances greater or equal than s'_1.



⌟

Parallel composition does not enjoy strong compositionality properties. Indeed in general $\widehat{|}(p_1, p_2, s_1, s_2)$ is related neither to s_1 nor to s_2, and even $\widehat{|}(p_1, p_2, s_1, \perp_{\mathbb{V}})$ is not related to s_1. Consider for instance the case where p_2 "consumes" a s_1 move.

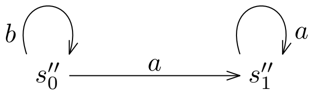
However, our metric domain $\mathbb{V}$ contains "contextual" information. We exploit this fact to show that a nice compositionality property, similar to non-extensivity [15], holds when the context and the distance are "compatible". A formal definition of compatibility follows.

▶ **Definition 25.** *A relation $\mathcal{R} \subseteq S \times P$ is a compatibility relation if, whenever $s \mathrel{\mathcal{R}} p$:*

1. $s \xrightarrow{\ell} s' \implies s' \mathrel{\mathcal{R}} p$;
2. $s \xrightarrow{\ell} s'$ *and* $p \xrightarrow{\ell} p' \implies s' \mathrel{\mathcal{R}} p'$ *and* $s \leq_{\mathbb{V}} s'$.

We say that s is p-compatible iff $s \mathrel{\mathcal{R}} p$ for some compatibility relation $\mathcal{R}$.

▶ **Example 26.** Consider again p_0, s_0, s_1 from Example 18 and $\delta(p_0|r_0, q_0|r_0)$ of Example 24. We have that s_0 is not p_0-compatible as Condition 2 from Definition 25 is violated: $s_0 \xrightarrow{a} s_1$ and $p_0 \xrightarrow{a} p_1$ but $s_0 \not\leq_{\mathbb{V}_0} s_1$. Instead, s''_0 below is p_0-compatible: it follows from the facts that s''_0 necessarily reduces to a greater or equal state, p_0 reduces to terminated states, which are vacuously compatible with every distance. Note that $\delta(p_0, q_0) = s_0 \leq_{\mathbb{V}_0} s''_0$ and $\delta(p_0|r_0, q_0|r_0) \leq_{\mathbb{V}_0} s''_0$. The second inclusion follows from the first by Proposition 27.

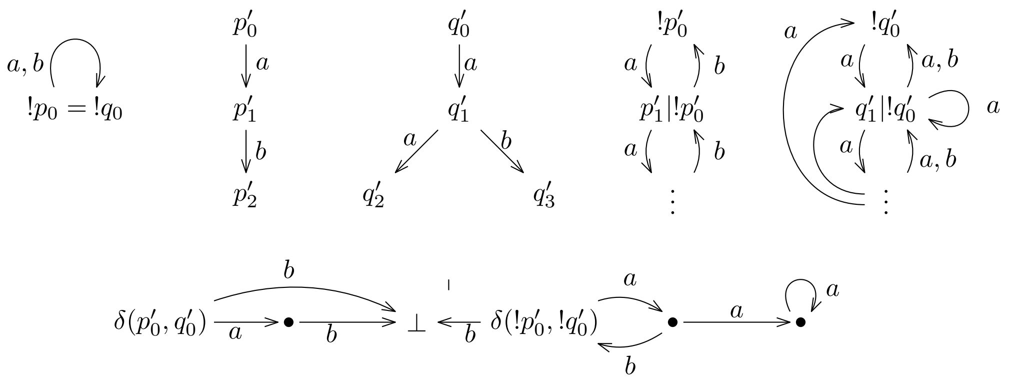


▶ **Proposition 27.** *If s_1 is p_2-compatible and s_2 is p_1-compatible, then $\widehat{|}(p_1, p_2, s_1, s_2) \leq_{\mathbb{V}} s_1 +_{\mathbb{V}} s_2$.*

5.5 Replication

We assume that P is closed both under operator $|$ (as defined in Section 5.4) and under $! : P \to P$, whose semantics is standard. In general, replication has bad compositionality properties: since it allows infinite behaviour, even a small distance in the parameter can get amplified to a much larger value. However, we show that $\widehat{!}$ is not expansive under the assumption that the parameter s always reduces to a larger or equal value and $+_{\mathbb{Q}}$ is idempotent. Such condition is of course quite strong, but it holds for instance when interpreting bisimilarity as a contextual bisimulation metric (see Section 4.1).

▶ **Example 28.** We have that $!p_0$ and $!q_0$ can both fire a a or b action and reduce to a process with the same behaviour (the simplest state with this property is drawn in the figure). Therefore, the distance $\delta(!p_0, !q_0) = \perp_{\mathbb{V}}$. In general, however, the distance among processes is not preserved by replication, as shown below:



⌟

▶ **Definition 29.** *We define* **Inc***, the set of increasing states, as the largest set $S' \subseteq S$ such that, whenever $s \in S'$ and $s \xrightarrow{\ell} s'$: $s \leq_{\mathbb{V}} s'$ and $s' \in S'$.*

▶ **Proposition 30.** *If s is increasing and $+_{\mathbb{Q}}$ is idempotent, then $\hat{!}(p, s) \leq_{\mathbb{V}} s$.*

6 Related Work & Conclusion

Quite a few works in the literature study context dependent relations. The closest to our work is the already mentioned study about environment paremetrized bisimilarity [23]. Our definition of CBM is similar to theirs, where the main differences are that we also consider quantitative aspects and that we explicitly work with a metric. The same work also provides an interesting logical characterisation of their relation in terms of Hennessy-Milner logic, but does not study compositionality. Since environment parametrized bisimilarity can be embedded into our framework, our compositionality results also hold for [23]. A closely related line of research [3, 20, 21] (non-exhaustive list) studies conditional bisimulations in an abstract categorical framework, where conditions are used to make assumptions on the environment. In particular, [21] introduces a notion of conditional bisimilarity for reactive systems and shows that conditional bisimilarity is a congruence. In [18], an early and a late notion of symbolic bisimilarity for value passing processes are introduced, where actual values are symbolically represented with boolean expressions with free variables. Symbolic bisimilarities are parametric w.r.t. a predicate that, in a sense, allows to make assumptions on the values that the context can send. Our notion of contextuality instead restricts the choices of the environment, and we do not consider explicit value passing.

Compositionality of behavioural metrics has been studied in the probabilistic setting [4, 8]. In [2], it has been shown that parallel composition is non-extensive. We remark that our notion of parallel composition is slightly more general than the one considered in [2], as in there processes necessarily synchronise on common actions. The work [14] studies compositionality for quite a few process algebraic operators, showing e.g. that non-deterministic sum is non-expansive, while parallel composition is non-extensive. The bang operator is shown Lipschitz continuous for the discounted metric, while not even uniformly continuous w.r.t. the non-discounted one. [16] introduces structural operational semantics formats that guarantee compositionality of operators. Basically, compositionality depends on how many parameters of the operator are copied from the source to the destination of the rules, weighted by probabilities and the discount factor.

Concluding Remarks. This paper introduces a new form of metric on the states of a LTS, called contextual behavioural metric, which enables contextual and quantitative reasoning. We study compositional properties of CBMs w.r.t. some operators, showing that, under the assumption that the immediate metric is non-extensive, the following hold: restriction is non-expansive, non-deterministic sum is non-extensive, prefixing enjoys a property slightly weaker than uniform continuity, parallel composition is non-extensive when the distance between components is compatible with the context and replication enjoys non-expansiveness under some (rather strong) assumptions on the underling quantale $\mathbb{Q}$.

Due to the generality of CBMs, our compositionality results extend to behavioural metrics as defined in Section 4.2. For instance, since the compatibility relation of Definition 25 holds trivially for the MLTS of behavioural metrics, we have that compositionality of parallel composition only depends on the compositionality of the immediate metric.

Our work is still preliminary, and indeed we are yet in the quest for an appropriate general notion of compositionality: here we tried to adapt concepts from the probabilistic setting [14, 16], where uniform continuity is considered as the most general notion of compositionality. In our setting not even prefixing enjoys uniform continuity, which should not come as a surprise, as quantales are not totally ordered in general. Our compositionality results have heterogeneous side conditions. Spelling out all the compositionality results in a uniform way would come with a high price: operators for which compositionality holds without any side condition, such as restriction, would have to be treated as those for which compositionality holds only modulo appropriate (and strong) hypotheses, such as replication. An interesting future work would be to infer the side conditions directly from SOS rules, or studying more operators or rule formats as in [16].

Another direction of future research would be to consider calculi with value and/or channel passing like the π-calculus: since strong bisimilarity is not a congruence in such settings, a promising approach could be a "contextualisation" of open-bisimilarity [32].

References

1 S. Abramsky. The lazy λ-calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

2 Giorgio Bacci, Giovanni Bacci, Kim G. Larsen, and Radu Mardare. Computing behavioral distances, compositionally. In *Proc. of MFCS*, volume 8087 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 2013.

3 Harsh Beohar, Barbara König, Sebastian Küpper, and Alexandra Silva. Conditional transition systems with upgrades. *Sci. Comput. Program.*, 186, 2020.

4 Konstantinos Chatzikokolakis, Daniel Gebler, Catuscia Palamidessi, and Lili Xu. Generalized bisimulation metrics. In *Proc. of CONCUR*, volume 8704 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2014.

5 Luca de Alfaro, Marco Faella, and Mariëlle Stoelinga. Linear and branching metrics for quantitative transition systems. In *Proc. of ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004.

6 Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. A semantic account of metric preservation. In *Proc. of POPL*, pages 545–556. ACM, 2017.

7 Josée Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labeled markov systems. In *Proc. of CONCUR*, volume 1664 of *LNCS*, pages 258–273. Springer, 1999.

8 Josée Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labelled markov processes. *Theor. Comput. Sci.*, 318(3):323–354, 2004.

9 Josée Desharnais, Radha Jagadeesan, Vineet Gupta, and Prakash Panangaden. The metric analogue of weak bisimulation for probabilistic processes. In *Proc. of LICS*, pages 413–422. IEEE Computer Society, 2002.

10 Josée Desharnais, François Laviolette, and Mathieu Tracol. Approximate analysis of probabilistic processes: Logic, simulation and games. In *Proc. of QEST*, pages 264–273. IEEE Computer Society, 2008.

11 Wenjie Du, Yuxin Deng, and Daniel Gebler. Behavioural pseudometrics for nondeterministic probabilistic systems. In *Proc. of SETTA*, volume 9984 of *Lecture Notes in Computer Science*, pages 67–84, 2016.

12 Wenjie Du, Yuxin Deng, and Daniel Gebler. Behavioural pseudometrics for nondeterministic probabilistic systems. *Sci. Ann. Comput. Sci.*, 32(2):211–254, 2022.

13 Norm Ferns, Prakash Panangaden, and Doina Precup. Metrics for finite markov decision processes. In *Proc. of AAAI*, pages 950–951. AAAI Press / The MIT Press, 2004.

14 Daniel Gebler, Kim G. Larsen, and Simone Tini. Compositional bisimulation metric reasoning with probabilistic process calculi. *Log. Methods Comput. Sci.*, 12(4), 2016.

15 Daniel Gebler, Kim Guldstrand Larsen, and Simone Tini. Compositional metric reasoning with probabilistic process calculi. In *Proc. of FoSSaCS*, volume 9034 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2015.

16 Daniel Gebler and Simone Tini. SOS specifications for uniformly continuous operators. *J. Comput. Syst. Sci.*, 92:113–151, 2018.

17 Guillaume Geoffroy and Paolo Pistone. A partial metric semantics of higher-order types and approximate program transformations. In *Proc. of CSL*, volume 183 of *LIPIcs*, pages 23:1–23:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

18 Matthew Hennessy and Huimin Lin. Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.

19 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

20 Mathias Hülsbusch and Barbara König. Deriving bisimulation congruences for conditional reactive systems. In *Proc. of FoSSaCS*, volume 7213 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2012.

21 Mathias Hülsbusch, Barbara König, Sebastian Küpper, and Lars Stoltenow. Conditional bisimilarity for reactive systems. In *Proc. of FSCD*, volume 167 of *LIPIcs*, pages 10:1–10:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

22 Ugo Dal Lago, Francesco Gavazzo, and Akira Yoshimizu. Differential logical relations, part I: the simply-typed case. In *Proc. of ICALP*, volume 132 of *LIPIcs*, pages 111:1–111:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

23 Kim Guldstrand Larsen. A context dependent equivalence between processes. *Theor. Comput. Sci.*, 49:184–215, 1987.

24 F. William Lawvere. Metric spaces, generalized logic, and closed categories. In *Rend. Sem. Mat. Fis. Milano*, volume 43, pages 135–166, 1973.

25 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

26 Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

27 David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, pages 167–183, 1981.

28 Paolo Pistone. On generalized metric spaces for the simply typed lambda-calculus. In *Proc. of LICS*, pages 1–14. IEEE, 2021.

29 Damien Pous and Davide Sangiorgi. *Enhancements of the bisimulation proof method*, pages 233–289. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011. `doi:10.1017/CBO9780511792588.007`.

30 Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. of ICFP*, pages 157–168. ACM, 2010.

31 Kimmo I. Rosenthal. *Quantales and their applications / Kimmo I. Rosenthal.* Pitman research notes in mathematics series ; 234. Longman Scientific & Technical, Essex, England, 1990.

32 Davide Sangiorgi. A theory of bisimulation for the pi-calculus. In *Proc. of CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 1993.

33 Franck van Breugel, Claudio Hermida, Michael Makkai, and James Worrell. An accessible approach to behavioural pseudometrics. In *Proc. of ICALP*, volume 3580 of *LNCS*, pages 1018–1030. Springer, 2005.

34 Franck van Breugel and James Worrell. Approximating and computing behavioural distances in probabilistic transition systems. *Theor. Comput. Sci.*, 360(1-3):373–385, 2006.

35 Rob J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theor. Comput. Sci.*, 177(2):329–349, 1997.

Priority Downward Closures

Ashwani Anand
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Georg Zetzsche
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

When a system sends messages through a lossy channel, then the language encoding all sequences of messages can be abstracted by its downward closure, i.e. the set of all (not necessarily contiguous) subwords. This is useful because even if the system has infinitely many states, its downward closure is a regular language. However, if the channel has congestion control based on priorities assigned to the messages, then we need a finer abstraction: The downward closure with respect to the priority embedding. As for subword-based downward closures, one can also show that these priority downward closures are always regular.

While computing finite automata for the subword-based downward closure is well understood, nothing is known in the case of priorities. We initiate the study of this problem and provide algorithms to compute priority downward closures for regular languages, one-counter languages, and context-free languages.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases downward closure, priority order, pushdown automata, non-deterministic finite automata, abstraction, computability

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.39

Related Version Some formal proofs and details can be found in the full version of the paper.
Full Version: `https://arxiv.org/abs/2307.07460` [2]

Funding Funded by the European Union (ERC, FINABIS, 101077902). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements The authors are grateful to Yousef Shakiba for discussions on the block downward closure of regular languages.

1 Introduction

When analyzing infinite-state systems, it is often possible to replace individual components by an overapproximation based on (subword) downward closures. Here, the *(subword) downward closure* of a language $L \subseteq \Sigma^*$ is the set of all words that appear as (not necessarily contiguous) subwords of members of L. This overapproximation is usually possible because the verified properties are not changed when we allow additional behaviors resulting from subwords. Furthermore, this overapproximation simplifies the system because a well-known result by Haines is that for *every language* $L \subseteq \Sigma^*$, its subword downward closure is regular.

This idea has been successfully applied to many verification tasks, such as the verification of restricted lossy channel systems [1], concurrent programs with dynamic thread spawning and bounded context-switching [3, 7], asynchronous programs (safety, termination, liveness [22], but also context-free refinement verification [8]), the analysis of thread pools [9], and safety of parameterized asynchronous shared-memory systems [25]. For these reasons, there has been a substantial amount of interest in algorithms to compute finite automata for subword downward closures of given infinite-state sytems [4–6, 12–14, 17, 18, 26–30].

34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 39; pp. 39:1–39:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

One situation where downward closures are useful is that of systems that send messages through a lossy channel, meaning that every message can be lost on the way. Then clearly, the downward closure of the set of sequences of messages is exactly the set of sequences observed by the receiver. This works as long as all messages can be dropped arbitrarily.

Priorities. However, if the messages are not dropped arbitrarily but as part of congestion control, then taking the set of all subwords would be too coarse an abstraction: Suppose we want to prioritize critical messages that can only be dropped if there are no lower-priority messages in the channel. For example, RFC 2475 describes an architecture that allows specifying relative priority among the IP packets from a finite set of priorities and allows the network links to drop lower priority packets to accommodate higher priority ones when the congestion in the network reaches a critical point [11]. As another example, in networks with an Asynchronous Transfer Mode layer, cells carry a priority in order to give preferences to audio or video packages over less time-critical packages [21]. In these situations, the subword downward closure would introduce behaviors that are not actually possible in the system.

To formally capture the effect of dropping messages by priorities, Haase, Schmitz and Schnoebelen [16] introduced *Priority Channel Systems (PCS)*. These feature an ordering on words (i.e. channel contents), called the *Prioritised Superseding Order (PSO)*, which allows the messages to have an assigned priority, such that higher priority messages can supersede lower priority ones. This order indeed allows the messages to be treated discriminatively, but the superseding is asymmetric. A message can be superseded only if there is a higher priority letter coming in the channel later. This means, PSO are the “priority counterpart” of the subword order for channels with priorities. In particular, in these systems, components can be abstracted by their *priority downward closure*, the downward closure with respect to the PSO. Fortunately, just as for subwords, priority downward closures are also always regular.

This raises the question of whether it is possible to compute finite automata for the priority downward closure for given infinite-state systems. For example, consider a recursive program that sends messages into a lossy channel with congestion control. Then, the set of possible message sequences that can arrive is exactly the priority downward closure $S{\downarrow_{\mathsf{P}}}$ of the language S of sent messages. Since S is context-free in this case, we would like to compute a finite automaton for $S{\downarrow_{\mathsf{P}}}$. While this problem is well-understood for subwords, nothing is known for priority downward closures.

Contribution. We initiate the study of computing priority downward closures. We show two main results. On the one hand, we study the setting above – computing priority downward closures of context-free languages. Here, we show that one can compute a doubly-exponential-sized automaton for its priority downward closure. On the other hand, we consider a natural restriction of context-free languages: We show that for one-counter automata, there is a polynomial-time algorithm to compute the priority downward closure.

Key technical ingredients. The first step is to consider a related order on words, which we call *block order*, which also has priorities assigned to letters, but imposes them more symmetrically. Moreover, we show that under mild assumptions, computing priority downward closures reduces to computing block downward closures.

Both our constructions – for one-counter automata and context-free languages – require new ideas. For one-counter automata, we modify the subword-based downward closures construction from [4] in a non-obvious way to block downward closures. Crucially, our modification relies on the insight that, in some word, repeating existing factors will always yield a word that is larger in the block order. For context-free languages, we present a novel inductive approach: We decompose the input language into finitely many languages with fewer priority levels and apply the construction recursively.

Outline of the paper. We fix notation in Section 2 and introduce the block order and show its relationship to the priority order in Section 3. In Sections 4–6, we then present methods for computing block and priority downward closures for regular languages, one-counter languages, and context-free languages, respectively.

2 Preliminaries

We will use the convention that $[i, j]$ denotes the set $\{i, i+1, \ldots, j\}$. By Σ, we represent a finite alphabet. Σ^* (Σ^+) denotes the set of (non-empty) words over Σ. When defining the priority order, we will equip Σ with a set of priorities with total order $(\mathcal{P}, \lessdot)$, i.e. there exists a fixed priority mapping from Σ to $\mathcal{P}$. The set of priority will be the set of integers $[0, d]$, with the canonical total order. By sets $\Sigma_{=p}$ ($p \in \mathcal{P}$), we denote the set of letters in Σ with priority p. For priority $p \in \mathcal{P}$, $\Sigma_{\leq p} = \Sigma_{=0} \cup \cdots \cup \Sigma_{=p}$, i.e. the set of letters smaller than or equal to p. For a word $w = a_0 a_1 \cdots a_k$, where $a_i \in \Sigma$, by $w[i, j]$, we denote the infix $a_i a_{i+1} \cdots a_{j-1} a_j$, and by $w[i]$, we denote a_i.

Finite automata and regular languages. A *non-deterministic finite state automaton (NFA)* is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*, Σ is its *input alphabet*, δ is its set of *edges* i.e. a finite subset of $Q \times \Sigma \cup \{\epsilon\} \times Q$, $q_0 \in Q$ is its *initial state*, and $F \subseteq Q$ is its set of *final states*. A word is accepted by $\mathcal{A}$ if it has a run from the initial state ending in a final state. The language *recognized* by an NFA $\mathcal{A}$ is called a regular language, and is denoted by $\mathcal{L}(\mathcal{A})$. The *size of a NFA*, denoted by $|\mathcal{A}|$, is the number of states in the NFA.

(Well-)quasi-orders. A *quasi-order*, denoted as $(X, \leq)$, is a set X with a reflexive and transitive relation $\leq$ on X. If $x \leq y$ (or equivalently, $y \geq x$), we say that x is smaller than y, or y is greater than x. If $\leq$ is also anti-symmetric, then it is called a *partial order*. If every pair of elements in X is comparable by $\leq$, then it is called a *total* or *linear* order. Let $(X, \leq_1)$ and $(Y, \leq_2)$ be two quasi orders, and $h : X \to Y$ be a function. We call h a *monomorphism* if it is one-to-one and $x_1 \leq_1 x_2 \iff h(x_1) \leq_2 h(x_2)$.

A quasi order $(X, \leq)$ is called a *well-quasi order (WQO)*, if any infinite sequence of elements $x_0, x_1, x_2, \ldots$ from X contains an increasing pair $x_i \leq x_j$ with $i < j$. If X is the set of words over some alphabet, then a WQO $(X, \leq)$ is called *multiplicative* if $\forall u, u', v, v' \in X$, $u \leq u'$ and $v \leq v'$ imply that $uv \leq u'v'$.

Subwords. For $u, v \in \Sigma^*$, we say $u \preccurlyeq v$, which we refer to as *subword order*, if u is a subword (not necessarily, contiguous) of v, i.e. if

$$\begin{aligned} u &= u_1 u_2 \cdots u_k \\ \text{and, } v &= v_0 u_1 v_1 u_2 v_2 \cdots v_{k-1} u_k v_k \end{aligned}$$

where $u_i \in \Sigma$ and $v_i \in \Sigma^*$. In simpler words, $u \preccurlyeq v$ if some letters of v can be dropped to obtain u. For example, let $\Sigma = [0, 1]$. Then, $0 \preccurlyeq 00 \preccurlyeq 010 \not\preccurlyeq 110$; 0 and 00 can be obtained by dropping letters from 00 and 010, respectively. But 010 cannot be obtained from 110, as the latter does not have sufficiently many 0s. If $u \preccurlyeq v$, we say that u is *subword smaller* than v, or simply that u is a *subword* of v. And we call a mapping from the positions in u to positions in v that witnesses $u \preccurlyeq v$ as the *witness position mapping*.

Since Σ is a WQO with the equality order, by Higman's lemma, Σ^* is a WQO with the subword order. It is in fact a multiplicative WQO: if $u \preccurlyeq u'$ and $v \preccurlyeq v'$, then dropping the same letters from $u'v'$ gives us uv.

Priority order. We take an alphabet Σ with priorities totally ordered by $\lessdot$. We say $u \preccurlyeq_{\mathsf{P}} v$, which we refer to as *priority order*, if $u = \epsilon$ or,

$$\begin{aligned} u &= u_1 u_2 \cdots u_k \\ \text{and, } v &= v_1 u_1 v_2 u_2 \cdots v_k u_k, \end{aligned}$$

such that $\forall i \in [1, k]$, $u_i \in \Sigma$ and $v_i \in \Sigma^*_{\leq u_i}$. It is easy to observe that the priority order is multiplicative, and is finer than the subword order, i.e. $\forall u, v \in \Sigma^*, u \preccurlyeq_{\mathsf{P}} v \implies u \preccurlyeq v$. As shown in [16, Theorem 3.6], the priority order on words over a finite alphabet with priorities is a well-quasi ordering:

▶ **Lemma 2.1.** *$(\Sigma^*, \preccurlyeq_{\mathsf{P}})$ is a WQO.*

Downward closure. We define the *subword downward closure* and *priority downward closure* for a language $L \subseteq \Sigma^*$ as follows:

$$L{\downarrow} := \{u \in \Sigma^* \mid \exists\, v \in L \colon u \preccurlyeq v\}, \qquad L{\downarrow}_{\mathsf{P}} := \{u \in \Sigma^* \mid \exists\, v \in L \colon u \preccurlyeq_{\mathsf{P}} v\}.$$

The following is the starting point for our investigation: It shows that for every language L, there exist finite automata for its downward closures w.r.t. $\preccurlyeq$ and $\preccurlyeq_{\mathsf{P}}$.

▶ **Lemma 2.2.** *Every subword downward closed sets and every priority downward closed set is regular.*

For the subword order, this was shown by Haines [19]. The same idea applies to the priority ordering: A downward closed set is the complement of an upward closed set. Therefore, and since every upward closed set in a well-quasi ordering has finitely many minimal elements, it suffices to show that the set of all words above a single word is a regular language. This, in turn, is shown using a simple automaton construction. In the full version, we prove an analogue of this for the block ordering (Lemma 3.5).

We stress that Lemma 2.2 is not effective: It does not guarantee that finite automata for downward closures can be computed for any given language. In fact, there are language classes for which they are not computable, such as reachability sets of *lossy channel systems* and *Church-Rosser languages* [15, 23]. Therefore, our focus will be on the question of how to effectively compute automata for priority downward closures.

3 The Block Order

We first define the block order formally and then give the intuition behind the definition. Let Σ be a finite alphabet, and $\mathcal{P} = [0, d]$ be a set of priorities with a total order $\lessdot$. Then for $u, v \in \Sigma^*$, where maximum priority occurring among u and v is p, we say $u \preccurlyeq_{\mathsf{B}} v$, if

i. if $u, v \in \Sigma^*_{=p}$, and $u \preccurlyeq v$, or

ii. if

$$\begin{aligned} u &= u_0 x_0 u_1 x_1 \cdots x_{n-1} u_n \\ \text{and, } v &= v_0 y_0 v_1 y_1 \cdots y_{m-1} v_m \end{aligned}$$

where $x_0, \ldots x_{n-1}, y_0, \ldots, y_{m-1} \in \Sigma_{=p}$, and for all $i \in [0, n]$, we have $u_i, v_i \in \Sigma^*_{\leq p-1}$ (the u_i and v_i are called *sub-p* blocks), and there exists a strictly monotonically increasing map $\phi : [0, n] \to [0, m]$, which we call the *witness block map*, such that

a. $u_i \preccurlyeq_{\mathsf{B}} v_{\phi(i)}$, $\forall i$,
b. $\phi(0) = 0$,
c. $\phi(n) = m$, and
d. $x_i \preccurlyeq v_{\phi(i)} y_{\phi(i)} v_{\phi(i)+1} \cdots v_{\phi(i+1)}$, $\forall i \in [0, n-1]$.

Intuitively, we say that u is *block smaller* than v, if either

- both words have letters of same priority, and u is a subword of v, or,
- the largest priority occurring in both words is p. Then we split both words along the priority p letters, to obtain sequences of sub-p blocks of words, which have words of strictly less priority. Then by item iia, we embed the sub-p blocks of u to those of v, such that they are recursively block smaller. Then with items iib and iic, we ensure that the first (and last) sub-p block of u is embedded in the first (resp., last) sub-p block of v. We will see later that this constraint allows the order to be multiplicative. Finally, by item iid, we ensure that the letters of priority p in u are preserved in v, i.e. every x_i indeed occurs between the embeddings of the sub-p block u_i and u_{i+1}.

▶ **Example 3.1.** Consider the alphabet $\Sigma = \{0^a, 0^b, 1^a, 1^b, 2^a, 2^b\}$ with priority set $\mathcal{P} = [0, 2]$ and $\Sigma_{=i} = \{i^a, i^b\}$. In the following examples, the color helps to identify the largest priority occurring in the words. First, notice that $\epsilon \preccurlyeq_{\mathsf{B}} 0^a \preccurlyeq_{\mathsf{B}} 0^a 0^b$, and hence

$$1^b 0^a \preccurlyeq_{\mathsf{B}} 0^a 1^b 0^a 0^a 1^a 0^a 0^b, \qquad \text{but} \qquad 1^b 0^a \not\preccurlyeq_{\mathsf{B}} 0^a 1^b 0^a 0^a 1^a 0^b 0^b.$$

This is because $0^a \not\preccurlyeq_{\mathsf{B}} 0^b 0^b$, i.e. the last sub-1 block of the former word cannot be mapped to the last sub-1 block of the latter word. As another example, we have

$$2^a 1^b 0^a \preccurlyeq_{\mathsf{B}} 0^a 2^a 0^a 1^b 0^a 0^a 1^a 0^a 0^b, \qquad \text{but} \qquad 2^a 1^b 0^a \not\preccurlyeq_{\mathsf{B}} 0^a 2^b 0^a 1^b 0^a 0^a 1^a 0^a 0^b.$$

This is because 2^a does not exist in the latter word, violating item iid. Finally, notice that

$$1^a 1^b \not\preccurlyeq_{\mathsf{B}} 1^a 2^a 1^b, \tag{1}$$

because the sub-2 block $1^a 1^b$ would have to be mapped to a single sub-2 block in the right-hand word; but none of them can accomodate $1^a 1^b$.

Note that by items iid and iia, we have that $u \preccurlyeq_{\mathsf{B}} v \implies u \preccurlyeq v$, for all $u, v \in \Sigma^*$. Then there exists a position mapping ρ from $[0, |u|]$ to $[0, |v|]$ such that $u[i] = v[\rho(i)]$, for all i. We say that a position mapping *respects block order* if for all i, $v[\rho(i), \rho(i+1)]$ contains letters of priorities smaller than $u[i]$ and $u[i+1]$. It is easy to observe that if $u \preccurlyeq_{\mathsf{B}} v$, then there exists a position mapping from u to v respecting the block order. The following is a straightforward repeated application of Higman's Lemma [20] (see the full version).

▶ **Theorem 3.2.** *$(\Sigma^*, \preccurlyeq_{\mathsf{B}})$ is a WQO.*

In fact, the block order is multiplicative, i.e. for all $u, v, u', v' \in \Sigma^*$ such that $u \preccurlyeq_{\mathsf{B}} u'$ and $v \preccurlyeq_{\mathsf{B}} v'$, it holds that $uv \preccurlyeq_{\mathsf{B}} u'v'$.

▶ **Lemma 3.3.** *$(\Sigma^*, \preccurlyeq_{\mathsf{B}})$ is a multiplicative WQO.*

Proof. For singleton $\mathcal{P}$, the result trivially holds because it coincides with the subword order. Let $(\Sigma^*_{\leq p-1}, \preccurlyeq_{\mathsf{B}})$ be multiplicative. Now we show that $(\Sigma^*_{\leq p}, \preccurlyeq_{\mathsf{B}})$ is multiplicative. To this end, let $u \preccurlyeq_{\mathsf{B}} u'$, $v \preccurlyeq_{\mathsf{B}} v'$, and ϕ, ψ be the witnessing block maps respectively. We assume

$$\begin{aligned}
u &= u_0 x_0 u_1 x_1 u_2 x_2 \cdots x_{k-1} u_k \\
v &= v_0 y_0 v_1 y_1 v_2 y_2 \cdots y_{l-1} v_l \\
u' &= u'_0 x'_0 u'_1 x'_1 u'_2 x'_2 \cdots x'_{k-1} u'_{k'} \\
v' &= v'_0 y'_0 v'_1 y'_1 v'_2 y'_2 \cdots y'_{l-1} v'_{l'}
\end{aligned}$$

where $x_i, y_i, x'_i, y'_i \in \Sigma_{=p}$. Consider the function $\delta\colon [0, k+l-1] \to [0, k'+l'-1]$ with

$$i \mapsto \begin{cases} \phi(i), & \text{if } 1 \leq i \leq k \\ \psi(i-k+1), & \text{if } k < i \leq k+l-1 \end{cases}$$

Since the k^{th} sub-p block of u and the 1^{st} sub-p block of v combines in uv to form one sub-p block, we have $k+l-1$ sub-p blocks. Similarly, $u'v'$ has $k'+l'-1$ sub-p blocks. And hence $u_k v_1 \preccurlyeq_{\mathsf{B}} u'_{k'} v'_1$, by induction hypothesis. The recursive embedding is obvious for other sub-p blocks. We also have that $\delta(0) = 0$ and $\delta(k+l-1) = k'+l'-1$. By monotonicity of ϕ and ψ, δ is also strictly monotonically increasing. Hence, δ witnesses $uv \preccurlyeq_{\mathsf{B}} u'v'$. ◀

Pumping. In the subword ordering, an often applied property is that for any words u, v, w, we have $uw \preccurlyeq uvw$, i.e. inserting any word leads to a superword. This is not true for the block ordering, as we saw in Example 3.1, (1). However, one of our key observations about the block order is the following property: If the word we insert is just a repetition of an existing factor, then this yields a larger word in the block ordering. This will be crucial for our downward closure construction for one-counter automata in Section 5.

▶ **Lemma 3.4** (Pumping Lemma). *For any $u, v, w \in \Sigma^*$, we have $uvw \preccurlyeq_{\mathsf{B}} uvvw$.*

Before we prove Lemma 3.4, let us note that by applying Lemma 3.4 multiple times, this implies that we can also repeat multiple factors. For instance, if $w = w_1w_2w_3w_4w_5$, then $w \preccurlyeq_{\mathsf{B}} w_1w_2^2w_3w_4^3w_5$. Figure 1 shows an example on how to choose the witness block map.

Proof. We proceed by induction on the number of priorities. If there is just a single priority (i.e. $\mathcal{P} = \{0\}$), then $\preccurlyeq_{\mathsf{B}}$ coincides with $\preccurlyeq$ and the statement is trivial. Let us assume the lemma is established for words with up to n priorities. We distinguish two cases.

- Suppose v contains only letters of priorities $[0, n]$. Then repeating v means repeating a factor inside a sub-$(n+1)$ block, which is a word with priorities in $[0, n]$. Hence, the statement follows by induction: Formally, this means we can use the embedding mapping that sends block i of uvw to block i of $uvvw$.
- Suppose v contains a letter of priority $n+1$. write $v = v_0x_1v_1 \cdots x_mv_m$, where $x_1, \ldots, x_m$ are the letters of priority $n+1$ in v and $v_0, \ldots, v_m$ are the sub-$(n+1)$ blocks of v. Then:

  $$uvw = uv_0x_1v_1 \cdots x_mv_mw, \qquad uvvw = uv_0x_1v_1 \cdots x_m \underbrace{v_mv_0x_1v_1 \cdots x_m}_{\text{skipped}} v_mw$$

  The idea is simple: Our witness block map just skips the m sub-$(n+1)$ blocks inside of $v_mv_0x_1 \cdots v_{m-1}x_m$. Thus, the sub-$(n+1)$ blocks in $uv_0x_1 \cdots v_{m-1}x_m$ are mapped to the same blocks in $uv_0x_1 \cdots v_{m-1}x_m$, and the sub-$(n+1)$ blocks in v_mw are mapped to the same blocks in v_mw. This is clearly a valid witness block map, since the first (resp. last) sub-$(n+1)$ block is mapped to the first (resp. last), and each sub-$(n+1)$ block is mapped to an identical sub-$(n+1)$ block. ◀

Regular downward closures. As for $\preccurlyeq$ and $\preccurlyeq_{\mathsf{P}}$, we define $L{\downarrow_{\mathsf{B}}} = \{u \in \Sigma^* \mid \exists v \in L\colon u \preccurlyeq_{\mathsf{B}} v\}$ for any $L \subseteq \Sigma^*$.

▶ **Lemma 3.5.** *For every $L \subseteq \Sigma^*$, $L{\downarrow_{\mathsf{B}}}$ is a regular language.*

For the proof of Lemma 3.5, one can argue as mentioned above: The complement $\Sigma^* \setminus (L{\downarrow_{\mathsf{B}}})$ of $L{\downarrow_{\mathsf{B}}}$ is upward closed. And since $\preccurlyeq_{\mathsf{B}}$ is a WQO, $\Sigma^* \setminus (L{\downarrow_{\mathsf{B}}})$ has finitely many minimal elements. It thus remains to show that for each word $w \in \Sigma^*$, the set of words $\preccurlyeq_{\mathsf{B}}$-larger than w is regular, which is a simple exercise. Details can be found in the full version.

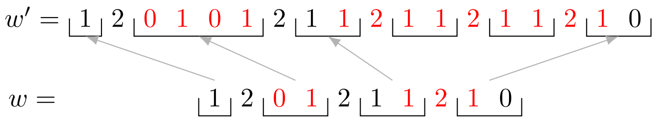

$w' =$ 1 2 0 1 0 1 2 1 1 2 1 1 2 1 1 2 1 0
$w =$ 1 2 0 1 2 1 1 2 1 0

Figure 1 Here $\Sigma = [0,2]$, $\mathcal{P} = [0,2]$, and $A_i = \{i\}$, $w = 12(01)21(121)0$ and $w' = 12(01)^2 21(121)^3 0$. The repeated segments are marked in red, and the arrows denote the witness block map.

Block order vs. priority order. We will later see (Theorem 4.4) that under mild conditions, computing priority downward closures reduces to computing block downward closures. The following lemma is the main technical ingredient in this: It shows that the block order refines the priority order on words that end in the same letter, assuming the alphabet has a certain shape. A priority alphabet $(\Sigma, \mathcal{P})$ with $\mathcal{P} = [1,d]$ is called *flat* if $|\Sigma_{=i}| = 1$ for each $i \in [1,d]$.

▶ **Lemma 3.6.** *If Σ is flat and $u, v \in \Sigma^* a$ for some $a \in \Sigma$, then $u \preccurlyeq_{\mathsf{B}} v$ implies $u \preccurlyeq_{\mathsf{P}} v$.*

Proof. Since $u \preccurlyeq_{\mathsf{B}} v$, there exists a witness position mapping ρ that maps the positions of the letters in u to that of v, such that it respects the block order, and it maps the last position of u to the last of v.

Let $u = u_0 u_1 \cdots u_k$. We say that a position mapping violates the priority order at position i (for $i \in [0, k-1]$), if $v[\rho(i)+1, \rho(i+1)]$ has a letter of priority higher than that of $u[i+1]$. Note that if ρ does not violate the priority order at any position, then $u \preccurlyeq_{\mathsf{P}} v$.

Let i be the largest position at which ρ violates the priority order, i.e. $v[\rho(i)+1, \rho(i+1)]$ has a letter of priority higher than that of $u[i+1]$. We show that if ρ respects the block order till position i, there exists another witness position mapping ρ' that respects the block order till position $i-1$, and has one few position of violation (i.e. no violation at position i).

We first observe that $u[i] > u[i+1]$, which holds since ρ respects the block order till position i, implying that $v[\rho(i)+1, \rho(i+1)]$ does not have a letter of priority higher than $min\{u[i], u[i+1]\}$, and if $u[i] \leq u[i+1]$, ρ does not violate the priority order at i.

Then observe that $v[\rho(i)+1, \rho(i+1)]$ does not have a letter with priority p, where $u[i] > p > u[i+1]$, otherwise the sub-$u[i]$ block of u immediately after $u[i]$, can not be embedded to that of v immediately after $v[\rho(i)]$, since it would have to be split along p, and the first sub-p block in v will not be mapped to any in u. Then $v[\rho(i)+1, \rho(i+1)]$ has letter of priority $u[i]$ (for a violation at i). Then consider the mapping ρ' that maps i to the last $u[i]$ letter in $v[\rho(i)+1, \rho(i+1)]$ (say at $v[j]$ for some j, $\rho(i)+1 \leq j \leq \rho(i+1)$).

This mapping respects the block order till position $i-1$, trivially, as we do not change the mapping before i. We show that there is no priority order violation at position i. This holds because the only larger priority letter occurring in $v[\rho(i)+1, \rho(i+1)]$ was $u[i]$, and due to the definition of ρ', $v[\rho'(i)+1, \rho'(i+1)]$ has no letter of priority higher than $u[i+1]$. Since we do not change the mapping after position i, ρ' does not introduce a violation at any position after i. Hence we have a new position mapping that has one few position of priority order violation. ◀

▶ Remark 3.7. We want to stress that the flatness assumption in Lemma 3.6 is crucial: Consider the alphabet Σ from the Example 3.1. Then $1^a 0^a \preccurlyeq_{\mathsf{B}} 1^a 1^b 0^a$, but $1^a 0^a \not\preccurlyeq_{\mathsf{P}} 1^a 1^b 0^a$. Here only one position mapping exists, and it is not possible to remap 1^a to 1^b since they are two distinct letters of same priority. Hence, we need to assume that each priority greater than zero has at most one letter.

4 Regular Languages

In this section, we show how to construct an NFA for the block downward closure of a regular language. To this end, we show that both orders are rational transductions.

Rational transductions. A *finite state transducer* is a tuple $\mathcal{A} = (Q, X, Y, E, q_0, F)$, where Q is a finite set of states, X and Y are *input* and *output alphabets*, respectively, E is the set of *edges* i.e. finite subset of $Q \times X^* \times Y^* \times Q$, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *final states*. A *configuration* of $\mathcal{A}$ is a triple $(q, u, v) \in Q \times X^* \times Y^*$. We write $(q, u, v) \rightarrow_{\mathcal{A}} (q', u', v')$, if there is an edge (q, x, y, q') with $u' = ux$ and $v' = vy$. If there is an edge (q, x, y, q'), we sometimes denote this fact by $q \xrightarrow{(x,y)}_{\mathcal{A}} q'$, and say "read x at q, output y, and goto q'". The *size of a transducer*, denoted by $|\mathcal{A}|$, is the number of its states.

A *transduction* is a subset of $X^* \times Y^*$ for some finite alphabets X, Y. The *transduction defined by* $\mathcal{A}$ is $\mathcal{T}(\mathcal{A}) = \{(u, v) \in X^* \times Y^* \mid (q_0, \epsilon, \epsilon) \rightarrow^*_{\mathcal{A}} (f, u, v)$ for some $f \in F\}$. A transduction is called *rational* if it is defined by some finite-state transducer. Sometimes we abuse the notation and output a regular language $R \subseteq Y^*$ on an edge, instead of a letter. It should be noted that this abuse is equivalent to original definition of finite state transducers.

We say that a language class $\mathcal{C}$ is *closed under rational transductions* if for each language $L \in \mathcal{C}$, and each rational transduction $R \subseteq X^* \times Y^*$, *the language obtained by applying the transduction* R to L, $RL \overset{def}{=} \{v \in Y^* \mid (u, v) \in R$ for some $u \in L\}$ also belongs to $\mathcal{C}$. We call such language classes *full trio*. Regular languages, context-free languages, recursively enumerable languages are some examples of full trios [10].

Transducers for orders. It is well-known that the subword order is a rational transduction, i.e. the relation $T = \{(u, v) \in X^* \times X^* \mid v \preccurlyeq u\}$ is defined by a finite-state transducer. For example, it can be defined by a one-state transducer that can non-deterministically decide to output or drop each letter. Note that on applying the transduction to any language, it gives the subword downward closure of the language. This means, for every $L \subseteq X^*$, we have $TL = L\!\downarrow$. We will now describe analogous transducers for the priority and block order.

▶ **Theorem 4.1.** *Given a priority alphabet with priorities* $[0, k]$*, one can construct in polynomial time a transducer for* $\preccurlyeq_{\mathsf{B}}$ *and a transducer for* $\preccurlyeq_{\mathsf{P}}$*, each of size* $\mathcal{O}(k)$*.*

Proof. The transducers for the block and priority order are similar. Intuitively, both remember the maximum of the priorities dropped or to be dropped, and keep or drop the coming letters accordingly. We show the transducer for the priority order here since it is applied in Theorem 4.4. The transducer for the block order is detailed in the full version.

Let Σ be a finite alphabet, with priorities $\mathcal{P} = [0, k]$. Consider the transducer that has one state for every priority, a non-final sink state, and a distinguished final state. If the transducer is in the state for priority r and reads a letter a of priority s, then

- if $s < r$, then it outputs nothing and stays in state r,
- if $s \geq r$, then it can output nothing, and go to state s,
- if $s \geq r$, it can also output a, and go to state 0, or the accepting state non-deterministically,
- for any other scenario, goes to the sink state.

The priority 0 state is the initial state. Intuitively, the transducer remembers the largest priority letter that has been dropped, and keeps only a letter of higher priority later. To be accepting, it has to read the last letter to go to the accepting final state. ◀

The following theorem states that the class of regular languages form a full trio.

▶ **Theorem 4.2** ([24, Corollary 3.5.5])**.** *Given an NFA $\mathcal{A}$ and a transducer $\mathcal{B}$, we can construct in polynomial time an NFA of size $|\mathcal{A}| \cdot |\mathcal{B}|$ for $\mathcal{T}(\mathcal{B})(\mathcal{L}(\mathcal{A}))$.*

Theorems 4.1 and 4.2 give us a polynomial size NFA recognizing the priority and block downward closure of a regular language, which is computable in polynomial time as well.

▶ **Theorem 4.3.** *Priority and block downward closures for regular languages are effectively computable in time polynomial in the number of states in the NFA recognizing the language.*

Theorem 4.3 and Lemma 3.6 now allow us to reduce the priority downward closure computability to computability for block order.

▶ **Theorem 4.4.** *If $\mathcal{C}$ is a full trio and we can effectively compute block downward closures for $\mathcal{C}$, then we can effectively compute priority downward closures.*

Proof. The key idea is to reduce priority downward closure computation to the setting where (i) all words end in the same letter and (ii) the alphabet is flat. Since by Lemma 3.6, on those languages, the block order is finer than the priority order, computing the block order will essentially be sufficient.

Let us first establish (i). Let $L \in \mathcal{C}$. Then for each $a \in \Sigma$, the language $L_a = L \cap \Sigma^* a$ belongs to $\mathcal{C}$. Since $L = \bigcup_{a\in\Sigma} L_a \cup E$ and thus $L{\downarrow_{\mathsf{P}}} = \bigcup_{a\in\Sigma} L_a{\downarrow_{\mathsf{P}}} \cup E$, it suffices to compute priority downward closures for each L_a, where $E = \{\epsilon\}$ if $\epsilon \in L$, else $\emptyset$. This means, it suffices to compute priority downward closures for languages where all words end in the same letter.

To achieve (ii), we make the alphabet flat. We say that $(\Sigma, \mathcal{P}')$ is the *flattening* of $(\Sigma, \mathcal{P} = [0, d])$, if $\mathcal{P}'$ is obtained by choosing a total order to Σ such that if a has smaller priority than b in $(\Sigma, \mathcal{P})$, then a has smaller priority than b in $(\Sigma, \mathcal{P}')$. (In other words, we pick an arbitrary linearization of the quasi-order on Σ that expresses "has smaller priority than"). Then, we assign priorities based on this total ordering. Let $\preccurlyeq_{\mathsf{B}}^{\mathsf{flat}}$ and $\preccurlyeq_{\mathsf{P}}^{\mathsf{flat}}$ denote the block order and priority order, resp., based on the flat priority assignment. It is a simple observation that for $u, v \in \Sigma^*$, we have that $u \preccurlyeq_{\mathsf{P}}^{\mathsf{flat}} v$ implies $u \preccurlyeq_{\mathsf{P}} v$.

Now observe that for $u, v \in L_a$, Lemma 3.6 tells us that $u \preccurlyeq_{\mathsf{B}}^{\mathsf{flat}} v$ implies $u \preccurlyeq_{\mathsf{P}}^{\mathsf{flat}} v$ and therefore also $u \preccurlyeq_{\mathsf{P}} v$. This implies that $(L_a{\downarrow_{\mathsf{B}}^{\mathsf{flat}}}){\downarrow_{\mathsf{P}}} = L_a{\downarrow_{\mathsf{P}}}$. By assumption, we can compute a finite automaton $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = L_a{\downarrow_{\mathsf{B}}^{\mathsf{flat}}}$. Since then $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{P}}} = (L_a{\downarrow_{\mathsf{B}}^{\mathsf{flat}}}){\downarrow_{\mathsf{P}}} = L_a{\downarrow_{\mathsf{P}}}$, we can compute $L_a{\downarrow_{\mathsf{P}}}$ by applying Theorem 4.3 to $\mathcal{A}$ to compute $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{P}}} = L_a{\downarrow_{\mathsf{P}}}$. ◀

5 One-counter Languages

In this section, we show that for the class of languages accepted by one-counter automata, which form a full-trio [10, Theorem 4.4], the block and priority downward closures can be computed in polynomial time. We prove the following theorem.

▶ **Theorem 5.1.** *Given an OCA $\mathcal{A}$, $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{B}}}$ and $\mathcal{L}(\mathcal{A}){\downarrow_{\mathsf{P}}}$ are computable in polynomial time.*

Here, the difficulty is that existing downward closure constructions exploit that inserting any letters in a word yields a super-word. However, for the block order, this might not be true: Introducing high-priority letters might split a block unintentionally. However, we observe that the subword closure construction from [4] can be modified so that when constructing larger runs (to show that our NFA only accepts words in the downward closure), we only repeat existing factors. Lemma 3.4 then yields that the resulting word is block-larger.

According to Theorem 4.4, it suffices to show that block downward closures are computable in polynomial time (an inspection of the proof of Theorem 4.4 shows that computing the priority downward closure only incurs a polynomial overhead).

One-counter automata. One-counter automata are finite state automata with a counter that can be incremented, decremented, or tested for zero. Formally, a *one-counter automaton (OCA)* $\mathcal{A}$ is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states, Σ is a finite alphabet and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \{-1, 0, +1, z\} \times Q$ is a set of transitions. Transitions $(p_1, a, s, p_2) \in \delta$ are classified as *incrementing* $(s = +1)$, *decrementing* $(s = -1)$, *internal* $(s = 0)$, or *test for zero*$(s = z)$.

A *configuration* of an OCA is a pair that consists of a state and a (non-negative) counter value, i.e., $(q, n) \in Q \times \mathbb{N}$. A sequence $\pi = (p_0, c_0), t_1, (p_1, c_1), t_2, \cdots, t_m, (p_m, c_m)$ where $(p_i, c_i) \in Q \times \mathbb{Z}$, $t_i \in \delta$ and $(p_{i-1}, c_{i-1}) \xrightarrow{t_i} (p_i, c_i)$ is called:

- a *quasi-run*, denoted $\pi = (p_0, c_0) \overset{w}{\underset{\mathcal{A}}{\Longrightarrow}} (p_m, c_m)$, if none of t_i is a test for zero;
- a *run*, denoted $\pi = (p_0, c_0) \xrightarrow{w}_{\mathcal{A}} (p_m, c_m)$, if all $(p_i, c_i) \in Q \times \mathbb{N}$.

For any quasi-run π as above, the sequence of transitions $t_1, \cdots, t_m$ is called a *walk* from the state p_0 to the state p_m. A run $(p_0, c_0) \xrightarrow{w} (p_m, c_m)$ is called *accepting* in $\mathcal{A}$ if $(p_0, c_0) = (q_0, 0)$ where q_0 is the initial state of $\mathcal{A}$ and p_m is a final state of $\mathcal{A}$, i.e. $p_m \in F$. In such a case, the word w is *accepted* by $\mathcal{A}$.

Simple one-counter automata. As we will show later, computing block downward closures of OCA easily reduces to the case of simple OCA. A *simple OCA (SOCA)* is defined analogously to OCA, with the differences that (i) there are no zero tests, (ii) there is only one final state, (iii) for acceptance, the final counter value must be zero.

We first show that the block downward closures can be effectively computed for the simple one-counter automata languages.

▶ **Proposition 5.2.** *Given a simple OCA $\mathcal{A}$, we can compute $\mathcal{L}(\mathcal{A})\downarrow_{\mathsf{B}}$ in polynomial time.*

We present a rough sketch of the construction, full details can be found in the full version. The starting point of the construction is the one for subwords in [4], but the latter needs to be modified in a non-obvious way using Lemma 3.4.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_f)$ be a simple OCA, with $|Q| = K$. We construct an NFA $\mathcal{B}$ that can simulate $\mathcal{A}$ in three different modes. In the first mode, it simulates $\mathcal{A}$ until the counter value reaches K, and when the value reaches $K + 1$, it switches to the second mode. The second mode simulates $\mathcal{A}$ while the counter value stays below $K^2 + K + 1$. Moreover, and this is where our construction differs from [4]: if $\mathcal{B}$ is in the second mode simulating $\mathcal{A}$ in some state q, then $\mathcal{B}$ can spontaneously execute a loop from q to q of $\mathcal{A}$ while ignoring its counter updates. When the counter value in the second mode drops to K again, $\mathcal{B}$ non-deterministically switches to the third mode to simulate $\mathcal{A}$ while the counter value stays below K. Thus, $\mathcal{B}$ only needs to track counter values in $[0, K^2 + K + 1]$, meaning they can be stored in its state. We claim that then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})\downarrow_{\mathsf{B}}$.

▶ **Lemma 5.3.** $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

If a word in $\mathcal{L}(\mathcal{A})$ has a run with counters bounded by $K^2 + K + 1$, then it trivially belongs to $\mathcal{L}(\mathcal{B})$. If the counters go beyond $K^2 + K + 1$, then with the classical "unpumping" argument, one can extract two loops, one increasing the counter, one decreasing it. These loops can then be simulated by the spontaneous loops in the second mode of $\mathcal{B}$.

The more interesting inclusion is the following:

▶ **Lemma 5.4.** $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})\downarrow_{\mathsf{B}}$.

We have to show that each spontaneous loop in $\mathcal{B}$ can be justified by padding the run with further loop executions so as to obtain a run of $\mathcal{A}$. This is possible because to execute such a spontaneous loop, we must have gone beyond K and later go to zero again. Thus, there exists a “pumping up” loop adding, say $k \geq 0$ to the counter, and a “pumping down” loop, subtracting, say $\ell \geq 0$ from the counter. We can therefore repeat all spontaneous loops so often that their effect – when seen as transitions in $\mathcal{A}$ – is a (positive or negative) multiple M of $k \cdot \ell$. Then, we execute the k- and the ℓ-loop so often so as to get the counter values so high that (i) our repeated spontaneous loops never cross zero and (ii) the effect difference of the new loops is exactly M. Since in our construction (in contrast to [4]), the padding only *repeated words that already exist* in the run of $\mathcal{B}$, Lemma 3.4 implies that the word of $\mathcal{B}$ embeds via the block order.

General OCA. Let us now show how to construct the block downward closure of general OCAs. Suppose we are given an OCA $\mathcal{A}$. For any two states p, q, consider the simple OCA $\mathcal{A}_{p,q}$ obtained from $\mathcal{A}$ by removing all zero tests, making p initial, and q final. Then $\mathcal{L}(\mathcal{A})$ is the set of words read from $(p, 0)$ to $(q, 0)$ without using zero tests. We now compute for each p, q a finite automaton $\mathcal{B}_{p,q}$ for the block downward closure of $\mathcal{A}_{p,q}$. Clearly, we may assume that $\mathcal{B}_{p,q}$ has exactly one initial state and one final state. Finally, we obtain the finite automaton $\mathcal{B}$ from $\mathcal{A}$ as follows: We remove all transitions *except* the zero tests. Each zero test from p to q is replaced with an edge $p \xrightarrow{\varepsilon} q$. Moreover, for any states p and q coming from $\mathcal{A}$, we glue in the automaton $\mathcal{B}_{p,q}$ (by connecting p with $\mathcal{B}_{p,q}$'s initial state and connecting $\mathcal{B}_{p,q}$'s final state with q). Then, since the block order is multiplicative, we have that $L(\mathcal{B})$ accepts exactly the block downward closure of $\mathcal{A}$.

Futhermore, note that since our construction for simple OCA is polynomial, the general case is as well: The latter employs the former to $|Q|^2$ simple OCAs.

6 Context-free Languages

The key trick in our construction for OCA was that we could modify the subword construction so that the overapproximating NFA $\mathcal{B}$ has the property that in any word from $\mathcal{L}(\mathcal{B})$, we can repeat factors to obtain a word from $\mathcal{A}$. This was possible because in an OCA, essentially any pair of loops – one incrementing, one decrementing – could be repeated to pad a run.

However, in context-free languages, the situation is more complicated. With a stack, any pumping must always ensure that stack contents match: It is not possible to compensate stack effects with just two loops. In terms of grammars, the core idea for subword closures of context-free languages L is usually to overapproximate “pump-like” derivations $X \overset{*}{\Rightarrow} uXv$ by observing that – up to subwords – they can generate any $u'Xv'$ where the letters of u' can occur on the left and the letters of v' can occur on the right in derivations $X \overset{*}{\Rightarrow} \cdot X \cdot$. Showing that all such words belong to the downward closure leads to derivations $X \overset{*}{\Rightarrow} u''\bar{v}Xv''\bar{u}$, where u'', v'' are super-words of u', v' such that $X \overset{*}{\Rightarrow} u''X\bar{u}$ and $X \overset{*}{\Rightarrow} \bar{v}Xv''$ can be derived. The additional infixes could introduce high priority letters and thus split blocks unintentionally.

Therefore, we provide a novel recursive approach to compute the block downward closure by decomposing derivations at high-priority letters. This is non-trivial as this decomposition might not match the decomposition given by derivation trees. Formally, we show:

▶ **Theorem 6.1.** *Given a context-free language $L \subseteq \Sigma^*_{\leq n}$, one can construct a doubly-exponential-sized automaton for $L{\downarrow_{\mathsf{B}}}$, and thus also for $L{\downarrow_{\mathsf{P}}}$.*

We do not know if this doubly exponential upper bound is optimal. A singly-exponential lower bound follows from the subword case: It is known that subword downward closures of context-free languages can require exponentially many states [6]. However, it is not clear whether for priority or block downward closures, there is a singly-exponential construction.

We again note that Theorem 4.4 (and its proof) imply that for Theorem 6.1, it suffices to compute a finite automaton for the block downward closure of the context-free language: Computing the priority downward closure then only increases the size polynomially.

Grammars. We present the construction using *context-free grammars*, which are tuples $\mathcal{G} = (N, T, P, S)$, where N is a finite set of *non-terminal letters*, T is a finite set of *terminal letters*, P is a finite set of *productions* of the form $X \to w$ with $X \in N$ and $w \in (N \cup T)^*$, and S is the *start symbol.* For $u, v \in (N \cup T)^*$, we have $u \Rightarrow v$ if there is a production $X \to w$ in P and $x, y \in (N \cup T)^*$ with $u = xXy$ and $v = xwy$. The *language generated by* $\mathcal{G}$, is then $\mathcal{L}(\mathcal{G}) := \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$, where $\overset{*}{\Rightarrow}$ is the reflexive, transitive closure of $\Rightarrow$.

Assumption on the alphabet. In order to compute block downward closures, it suffices to do this for flat alphabets (see Section 3). The argument is essentially the same as in Theorem 4.4: By flattening the alphabet as in the proof of Theorem 4.4, we obtain a finer block order, so that first computing an automaton for the flat alphabet and then applying Theorem 4.3 to the resulting finite automaton will yield a finite automaton for the original (non-flat) alphabet. In the following, we will assume that the input grammar $\mathcal{G}$ is in Chomsky normal form, meaning every production is of the form $X \to YZ$ for non-terminals X, Y, Z, or of the form $X \to a$ for a non-terminal X and a terminal a.

Kleene grammars. Suppose we are given a context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$. Roughly speaking, the idea is to construct another grammar $\mathcal{G}'$ whose language has the same block downward closure as $\mathcal{L}(\mathcal{G})$, but with the additional property that every word can be generated using a derivation tree that is *acyclic*, meaning that each path contains every non-terminal at most once. Of course, if this were literally true, $\mathcal{G}'$ would generate a finite language. Therefore, we allow a slightly expanded syntax: We allow Kleene stars in context-free productions.

This means, we allow right-hand sides to contain occurrences of B^*, where B is a non-terminal. The semantics is the obvious one: When applying such a rule, then instead of inserting B^*, we can generate any B^k with $k \geq 0$. We call grammars with such productions *Kleene grammar.* A *derivation tree* in a Kleene grammar is defined as for context-free grammars, aside from the expected modification: If some B^* occurs on a right-hand side, then we allow any (finite) number of B-labeled children in the respective place. Then indeed, a Kleene grammar can generate infinite sets using acyclic derivation trees. Given a Kleene grammar $\mathcal{H}$, let $\mathsf{acyclic}(\mathcal{H})$ be the set of words generated by $\mathcal{H}$ using acyclic derivation trees.

▶ **Lemma 6.2.** *Given a Kleene grammar $\mathcal{H}$, one can construct an exponential-sized finite automaton accepting* $\mathsf{acyclic}(\mathcal{H})$*.*

Proof sketch. The automaton simulates a (say, preorder) traversal of an acyclic derivation tree of $\mathcal{H}$. This means, its state holds the path to the currently visited node in the derivation tree. Since every path has length at most $|N|$, where N is the set of non-terminals of $\mathcal{H}$, the automaton has at most exponentially many states. ◀

Given Lemma 6.2, for Theorem 6.1, it suffices to construct a Kleene grammar $\mathcal{G}'$ of exponential size such that $\mathsf{acyclic}(\mathcal{G}')\downarrow_{\mathsf{B}} = \mathcal{L}(\mathcal{G})\downarrow_{\mathsf{B}}$.

Normal form and grammar size. We will ensure that in the constructed grammars, the productions are of the form (i) $X \to w$, where w is a word of length ≤ 3 and consisting of non-terminals Y or Kleene stars Y^* or (ii) $X \to a$ where a is a terminal. This means, the total size of the grammar is always polynomial in the number of non-terminals. Therefore, to analyze the complexity, it will suffice to measure the number of non-terminals.

Highest occurring priorities. Similar to classical downward closure constructions for context-free languages, we want to overapproximate the set of words generated by "pump derivations" of the form $X \overset{*}{\Rightarrow} uXv$. Since we are dealing with priorities, we first partition the set of such derivations according to the highest occurring priorities, on the left and on the right. Thus, for $r, s \in [0, p]$, we will consider all derivations $X \overset{*}{\Rightarrow} uXv$ where r is the highest occurring priority in u and s is the highest occurring priority in v. To ease notation, we define $\Sigma_{\max r}$ to be the set of words in $\Sigma^*_{\leq r}$ in which r is the highest occurring priority. Since $\Sigma_{\max r} = \Sigma^+_{\max r}$, we will write $\Sigma^+_{\max r}$ to remind us that this is not an alphabet. Notice that for $r \in [1, p]$, we have $\Sigma^+_{\max r} = \Sigma^*_{\leq r} r \Sigma^*_{\leq r}$ and $\Sigma^+_{\max 0} = \Sigma^*_{\leq 0}$.

Language of ends. In order to perform an inductive construction, we need a way to transform pairs $(u, v) \in \Sigma^+_{\max r} \times \Sigma^+_{\max s}$ into words over an alphabet with fewer priorities. Part of this will be achieved by the *end maps* $\overleftarrow{\tau}_r(\cdot)$ and $\overrightarrow{\tau}_s(\cdot)$ as follows. Let $\hat{\Sigma}$ be the priority alphabet obtained from Σ by adding the letters $\#$, $\overleftarrow{\#}$, and $\overrightarrow{\#}$ as letters with priority zero. Now for $r \in [1, p]$, the function $\overleftarrow{\tau}_r \colon \Sigma^+_{\max r} \to \hat{\Sigma}^*_{\leq r-1}$ is defined as:

$$\overleftarrow{\tau}_r(w) = u\overleftarrow{\#}v, \text{ where } w = urx_1r \cdots x_nrv \text{ for some } n \geq 0,\ u, v, x_1, \ldots, x_n \in \Sigma^*_{\leq r-1}.$$

Thus, $\overleftarrow{\tau}_r(w)$ is obtained from w by replacing the largest possible infix surrounded by r with $\overleftarrow{\#}$. For $r = 0$, it will be convenient to have the constant function $\overleftarrow{\tau}_0 \colon \Sigma^+_{\max 0} \to \{\overleftarrow{\#}\}$. Analogously, we define for $s \in [1, p]$ the function $\overrightarrow{\tau}_s \colon \Sigma^+_{\max s} \to \hat{\Sigma}^*_{\leq s-1}$ by

$$\overrightarrow{\tau}_s(w) = u\overrightarrow{\#}v, \text{ where } w = usx_1s \cdots x_nsv \text{ for some } n \geq 0,\ u, v, x_1, \ldots, x_n \in \Sigma^*_{\leq s-1}.$$

Moreover, we also set $\overrightarrow{\tau}_0 \colon \Sigma^+_{\max 0} \to \{\overrightarrow{\#}\}$ to be the constant function yielding $\overrightarrow{\#}$.

In particular, for $r, s \in [1, p]$, we have $\overleftarrow{\tau}_r(w), \overrightarrow{\tau}_s(w) \in \hat{\Sigma}_{\leq p-1}$ and thus we have reduced the number of priorities. Now consider for $r, s \in [0, p]$ the language

$$E_{X,r,s} = \{\overleftarrow{\tau}_r(u)\#\overrightarrow{\tau}_s(v) \mid X \overset{*}{\Rightarrow} uXv,\ u \in \Sigma^*_{\leq r} r \Sigma^*_{\leq r},\ v \in \Sigma^*_{\leq s} s \Sigma^*_{\leq s}\}.$$

For the language $E_{X,r,s}$, it is easy to construct a context-free grammar:

▶ **Lemma 6.3.** *Given $\mathcal{G}$, a non-terminal X, and $r, s \in [0, p]$, one can construct a grammar $\mathcal{E}_{X,r,s}$ for $E_{X,r,s}$ of linear size.*

Defining the sets $E_{X,r,s}$ with fresh zero-priority letters $\#$, $\overleftarrow{\#}$, $\overrightarrow{\#}$ is a key trick in our construction: Note that each word in $E_{X,r,s}$ is of the form $u\overleftarrow{\#}v\#w\overrightarrow{\#}x$ for $u, v, w, x \in \Sigma^*_{\leq p-1}$. The segments u, v, w, x come from different blocks of the entire generated word, so applying the block downward closure construction recursively to $E_{X,r,s}$ must guarantee that these segments embed as if they were blocks. However, there are only a bounded number of segments. Thus, we can reduce the number of priorities while retaining the block behavior by using fresh zero-priority letters. This is formalized in the following lemma:

▶ **Lemma 6.4.** *For $u, u', v, v' \in \Sigma^*_{\leq p}$, we have $u\#v \preccurlyeq_{\mathsf{B}} u'\#v'$ iff both (i) $u \preccurlyeq_{\mathsf{B}} u'$ and (ii) $v \preccurlyeq_{\mathsf{B}} v'$.*

Language of repeated words. Roughly speaking, the language $E_{X,r,s}$ captures the "ends" of words derived in derivations $X \overset{*}{\Rightarrow} uXv$ with $u \in \Sigma^+_{\max r}$ and $v \in \Sigma^+_{\max s}$: On the left, it keeps everything that is not between two occurrences of r and on the right, it keeps everything not between two occurrences of s. We now need languages that capture the infixes that can occur between r's and s's, respectively. Intuitively, these are the words that can occur again and again in words derived from X. There is a "left version" and a "right version". We set for $r, s \in [1, p]$:

$$\overleftarrow{R}_{X,r,s} = \{yr \mid y \in \Sigma^*_{\le r-1},\ \exists x, z \in \Sigma^*_{\le r},\ v \in \Sigma^+_{\max s} \colon X \overset{*}{\Rightarrow} xryrzXv\}$$

$$\overrightarrow{R}_{X,r,s} = \{ys \mid y \in \Sigma^*_{\le s-1},\ \exists u \in \Sigma^+_{\max r},\ x, z \in \Sigma^*_{\le r} \colon X \overset{*}{\Rightarrow} uXxsysz\}.$$

The case where one side has highest priority zero must be treated slightly differently: There are no enveloping occurrences of some $r, s \in [1, p]$. However, we can overapproximate those words by the set of all words over a particular alphabet. Specifically, for $r, s \in [0, p]$, we set

$$\overrightarrow{R}_{X,0,s} = \{a \in \Sigma_{\le 0} \mid \exists u \in \Sigma^+_{\max 0},\ v \in \Sigma^+_{\max s} \colon X \overset{*}{\Rightarrow} uXv,\ a \text{ occurs in } u\}$$

$$\overleftarrow{R}_{X,r,0} = \{a \in \Sigma_{\le 0} \mid \exists u \in \Sigma^+_{\max r},\ v \in \Sigma^+_{\max 0} \colon X \overset{*}{\Rightarrow} uXv,\ a \text{ occurs in } v\}$$

▶ **Lemma 6.5.** *Given $\mathcal{G}$, a non-terminal X, and $r, s \in [0, p]$, one can construct grammars $\overleftarrow{\mathcal{R}}_{X,r,s}$, $\overrightarrow{\mathcal{R}}_{X,r,s}$ for $\overleftarrow{R}_{X,r,s}$,$\overrightarrow{R}_{X,r,s}$, respectively, of linear size.*

Overapproximating derivable words. The languages $E_{X,r,s}$ and $\overleftarrow{R}_{X,r,s}$ and $\overrightarrow{R}_{X,r,s}$ now serve to define overapproximations of the set of $(u, v) \in \Sigma^+_{\max r} \times \Sigma^+_{\max s}$ with $X \overset{*}{\Rightarrow} uXv$: One can obtain each such pair by taking a word from $E_{X,r,s}$, replacing $\overleftarrow{\#}$ and $\overrightarrow{\#}$, resp., by words in $r\overleftarrow{R}^*_{X,r,s}$ ($\overleftarrow{R}^*_{X,0,s}$ if $r = 0$) and $s\overrightarrow{R}^*_{X,r,s}$ ($\overrightarrow{R}^*_{X,r,0}$ if $s = 0$), respectively. By choosing the right words from $E_{X,r,s}$, $\overleftarrow{R}_{X,r,s}$, and $\overrightarrow{R}_{X,r,s}$, we can thus obtain $u\#v$. However, this process will also yield other words that cannot be derived. However, the key idea in our construction is that every word obtainable in this way from $E_{X,r,s}$, $\overleftarrow{R}_{X,r,s}$, and $\overrightarrow{R}_{X,r,s}$ will be in the block downward closure of a pair of words derivable using $X \overset{*}{\Rightarrow} \cdot X \cdot$.

Let us make this precise. To describe the set of words obtained from $E_{X,r,s}$, $\overleftarrow{R}_{X,r,s}$, and $\overrightarrow{R}_{X,r,s}$, we need the notion of a substitution. For alphabets Γ_1, Γ_2, a *substitution* is a map $\sigma \colon \Gamma_1 \to 2^{\Gamma_2^*}$ that yields a language in Γ_2 for each letter in Γ_1. Given a word $w = w_1 \cdots w_n$ with $w_1, \dots w_n \in \Gamma_1$, we define $\sigma(w) := \sigma(w_1) \cdots \sigma(w_n)$. Then for $K \subseteq \Gamma_1^*$, we set $\sigma(K) = \bigcup_{w \in K} \sigma(w)$. Now let $\Sigma_{X,r,s} \colon \hat{\Sigma}_{\le p} \to 2^{\hat{\Sigma}^*_{\le p}}$ be the substitution that maps every letter in $\Sigma_{\le p} \cup \{\#\}$ to itself (as a singleton) and maps $\overleftarrow{\#}$ to $r\overleftarrow{R}^*_{X,r,s}$ and $\overrightarrow{\#}$ to $s\overrightarrow{R}^*_{X,r,s}$. Now our observation from the previous paragraph can be phrased as:

▶ **Lemma 6.6.** *For every $u\#v \in \Sigma_{X,r,s}(E_{X,r,s})$, there are $u' \in \Sigma^+_{\max r}$ and $v' \in \Sigma^+_{\max s}$ with $u \preccurlyeq_{\mathsf{B}} u'$, $v \preccurlyeq_{\mathsf{B}} v'$, and $X \overset{*}{\Rightarrow} u'Xv'$.*

Constructing the Kleene grammar. We now construct the Kleene grammar for $\mathcal{L}(\mathcal{G})\downarrow_{\mathsf{B}}$ by first computing the grammars $\mathcal{E}_{X,r,s}$, $\overleftarrow{\mathcal{R}}_{X,r,s}$, and $\overrightarrow{\mathcal{R}}_{X,r,s}$ for each non-terminal X and each $r, s \in [1, p]$. Then, since $\mathcal{E}_{X,r,s}$, $\overleftarrow{\mathcal{R}}_{X,r,s}$, and $\overrightarrow{\mathcal{R}}_{X,r,s}$ generate languages with at most $p - 1$ priorities, we can call our construction recursively to obtain grammars $\mathcal{E}'_{X,r,s}$, $\overleftarrow{\mathcal{R}}'_{X,r,s}$, and $\overrightarrow{\mathcal{R}}'_{X,r,s}$, respectively. Then, we add all productions of the grammars $\mathcal{E}'_{X,r,s}$, $\overleftarrow{\mathcal{R}}'_{X,r,s}$, and

$\overrightarrow{\mathcal{R}}'_{X,r,s}$ to $\mathcal{G}'$. Moreover, we make the following modifications: Each production of the form $Y \to \overleftarrow{\#}$ (resp. $Y \to \overrightarrow{\#}$) in $\mathcal{E}_{X,r,s}$ is replaced with $Y \to Z_r \overleftarrow{S}^*_{X,r,s}$ (resp. $Y \to Z_s \overrightarrow{S}^*_{X,r,s}$), where $\overleftarrow{S}_{X,r,s}$ (resp. $\overrightarrow{S}_{X,r,s}$) is the start symbol of $\overleftarrow{\mathcal{R}}'_{X,r,s}$ (resp. $\overrightarrow{\mathcal{R}}'_{X,r,s}$), and Z_r is a fresh non-terminal used to derive r or ε: We also have $Z_r \to r$ for each $r \in [1,p]$ and $Z_0 \to \varepsilon$. Moreover, each production $Y \to \#$ in $\mathcal{E}'_X$ is removed and replaced with a production $Y \to w$ for each production $X \to w$ in $\mathcal{G}$. We call the resulting grammar $\mathcal{G}'$.

Correctness. Let us now observe that the grammar $\mathcal{G}'$ does indeed satisfy $\mathcal{L}(\mathcal{G}'){\downarrow_{\mathsf{B}}} = \mathcal{L}(\mathcal{G}){\downarrow_{\mathsf{B}}}$. The inclusion "$\supseteq$" is trivial as $\mathcal{G}'$ is obtained by adding productions. For the converse, we need some terminology. We say that a derivation tree t_1 in $\mathcal{G}'$ is obtained using an *expansion step* from t_0 if we take an X-labeled node x in t_0, where X is a non-terminal from $\mathcal{G}$, and replace this node by a derivation $X \overset{*}{\Rightarrow} uwv$ using newly added productions (i.e. using $\mathcal{E}_{X,r,s}$, $\overleftarrow{\mathcal{R}}_{X,r,s}$, and $\overrightarrow{\mathcal{R}}_{X,r,s}$ and some $Y \to w$ where $X \to w$ was the production applied to x in t_0). Then by construction of $\mathcal{G}'$, any derivation in $\mathcal{G}'$ can be obtained from a derivation in $\mathcal{G}$ by finitely many expansion steps. An induction on the number of expansion steps shows:

▶ **Lemma 6.7.** *We have* $\mathcal{L}(\mathcal{G}'){\downarrow_{\mathsf{B}}} = \mathcal{L}(\mathcal{G}){\downarrow_{\mathsf{B}}}$.

Acyclic derivations suffice. Now that we have the grammar $\mathcal{G}'$ with $\mathcal{L}(\mathcal{G}'){\downarrow_{\mathsf{B}}} = \mathcal{L}(\mathcal{G}){\downarrow_{\mathsf{B}}}$, it remains to show that every word in $\mathcal{G}'$ can be derived using an acyclic derivation:

▶ **Lemma 6.8.** $\mathsf{acyclic}(\mathcal{G}'){\downarrow_{\mathsf{B}}} = \mathcal{L}(\mathcal{G}){\downarrow_{\mathsf{B}}}$.

Essentially, this is due to the fact that any repetition of a non-terminal X on some path means that we can replace a corresponding derivation $X \overset{*}{\Rightarrow} uXv$ by using new productions from $\mathcal{E}'_{X,r,s}$, $\overleftarrow{\mathcal{R}}'_{X,r,s}$, and $\overrightarrow{\mathcal{R}}'_{X,r,s}$. Since these also have the property that every derivation can be made acyclic, the lemma follows. See the full version for details.

Complexity analysis. To estimate the size of the constructed grammar, let $f_p(n)$ be the maximal number of non-terminals of a constructed Kleene grammar for an input grammar with n non-terminals over p priorities. By Lemmas 6.3 and 6.5, there is a constant c such that each grammar $\mathcal{E}_X$, $\overleftarrow{\mathcal{R}}_X$, and $\overrightarrow{\mathcal{R}}_X$ has at most cn non-terminals. Furthermore, $\mathcal{G}'$ is obtained by applying our construction to $3n(p+1)^2$ grammars with $p-1$ priorities of size cn, and adding Z_p. Thus $f_p(n) \leq n + 3n(p+1)^2 f_{p-1}(cn) + 1$. Since $f_{p-1}(n) \geq 1$, we can simplify to $f_p(n) \leq 4n(p+1)^2 f_{p-1}(cn)$. It is easy to check that $f_0(n) \leq 4n+1 \leq 5n$, because $\mathcal{E}_{X,0,0}$ and $\overleftarrow{\mathcal{R}}_{X,0,0}$ and $\overrightarrow{\mathcal{R}}_{X,0,0}$ each only have one non-terminal. Hence $f_p(n) \leq (4n(p+1)^2)^p f_0(c^p n) \leq (4n(p+1)^2) \cdot 4(c^p n)$, which is exponential in the size of $\mathcal{G}$.

7 Conclusion

We have initiated the study of computing priority and block downward closures for infinite-state systems. We have shown that for OCA, both closures can be computed in polynomial time. For CFL, we have provided a doubly exponential construction.

Many questions remain. First, we leave open whether the doubly exponential bound for context-free languages can be improved to exponential. An exponential lower bound is easily inherited from the exponential lower bound for subwords [6]. Moreover, it is an intriguing question whether computability of subword downward closures for vector addition systems [17], higher-order pushdown automata [18], and higher-order recursion schemes [12] can be strengthened to block and priority downward closures.

References

1 Parosh Aziz Abdulla, Luc Boasson, and Ahmed Bouajjani. Effective lossy queue languages. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 639–651. Springer, 2001. doi:10.1007/3-540-48224-5_53.

2 Ashwani Anand and Georg Zetzsche. Priority downward closures, 2023. arXiv:2307.07460.

3 Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Log. Methods Comput. Sci.*, 7(4), 2011. doi:10.2168/LMCS-7(4:4)2011.

4 Mohamed Faouzi Atig, Dmitry Chistikov, Piotr Hofman, K. Narayan Kumar, Prakash Saivasan, and Georg Zetzsche. The complexity of regular abstractions of one-counter languages. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 207–216, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2933575.2934561.

5 Mohamed Faouzi Atig, Roland Meyer, Sebastian Muskalla, and Prakash Saivasan. On the upward/downward closures of Petri nets. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 – Aalborg, Denmark*, volume 83 of *LIPIcs*, pages 49:1–49:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.49.

6 Georg Bachmeier, Michael Luttenberger, and Maximilian Schlund. Finite automata for the sub- and superword closure of cfls: Descriptional and computational complexity. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications – 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 473–485. Springer, 2015. doi:10.1007/978-3-319-15579-1_37.

7 Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded analysis of concurrent programs (invited talk). In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 3:1–3:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ICALP.2023.3.

8 Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of context-free specifications. *Proc. ACM Program. Lang.*, 7(POPL):2141–2170, 2023. doi:10.1145/3571266.

9 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of thread pools. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:10.1145/3498678.

10 J. Berstel. *Transductions and Context-Free Languages*. Vieweg+Teubner Verlag, 1979.

11 S. Blake, D. Black, M. Carlson, Elwyn B. Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. *RFC*, 2475:1–36, 1998.

12 Lorenzo Clemente, Pawel Parys, Sylvain Salvati, and Igor Walukiewicz. The diagonal problem for higher-order recursion schemes is decidable. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 96–105. ACM, 2016. doi:10.1145/2933575.2934527.

13 Bruno Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, January 1991.

14 Jean Goubault-Larrecq, Simon Halfon, Prateek Karandikar, K. Narayan Kumar, and Philippe Schnoebelen. The ideal approach to computing closed subsets in well-quasi-ordering. *CoRR*, abs/1904.10703, 2019. arXiv:1904.10703.

15 Hermann Gruber, Markus Holzer, and Martin Kutrib. The size of higman–haines sets. *Theoretical Computer Science*, 387(2):167–176, 2007. Descriptional Complexity of Formal Systems. doi:10.1016/j.tcs.2007.07.036.

16 Christoph Haase, Sylvain Schmitz, and Philippe Schnoebelen. The Power of Priority Channel Systems. *Logical Methods in Computer Science*, Volume 10, Issue 4, December 2014. doi:10.2168/LMCS-10(4:4)2014.

17 Peter Habermehl, Roland Meyer, and Harro Wimmel. The downward-closure of Petri net languages. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*, volume 6199 of *Lecture Notes in Computer Science*, pages 466–477. Springer, 2010. doi:10.1007/978-3-642-14162-1_39.

18 Matthew Hague, Jonathan Kochems, and C.-H. Luke Ong. Unboundedness and downward closures of higher-order pushdown automata. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20–22, 2016*, pages 151–163. ACM, 2016. doi:10.1145/2837614.2837627.

19 Leonard H. Haines. On free monoids partially ordered by embedding. *Journal of Combinatorial Theory*, 6(1):94–98, 1969. doi:10.1016/S0021-9800(69)80111-0.

20 Graham Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathematical Society*, s3-2(1):326–336, January 1952. doi:10.1112/plms/s3-2.1.326.

21 Jean-Yves Le Boudec. The asynchronous transfer mode: a tutorial. *Computer Networks and ISDN Systems*, 24(4):279–309, 1992. The ATM-Asynchronous Transfer Mode. doi:10.1016/0169-7552(92)90114-6.

22 Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. General decidability results for asynchronous shared-memory programs: Higher-order and beyond. *Log. Methods Comput. Sci.*, 18(4), 2022. doi:10.46298/lmcs-18(4:2)2022.

23 Richard Mayr. Undecidable problems in unreliable computations. In Gaston H. Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, pages 377–386, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

24 Jeffrey Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008. doi:10.1017/CBO9780511808876.

25 Salvatore La Torre, Anca Muscholl, and Igor Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 72–84. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.CONCUR.2015.72.

26 Jan van Leeuwen. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics*, 21(3):237–252, 1978.

27 Georg Zetzsche. An approach to computing downward closures. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming – 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2015. doi:10.1007/978-3-662-47666-6_35.

28 Georg Zetzsche. Computing downward closures for stacked counter automata. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, volume 30 of *LIPIcs*, pages 743–756. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.STACS.2015.743.

29 Georg Zetzsche. The complexity of downward closure comparisons. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 123:1–123:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.123.

30 Georg Zetzsche. Separability by piecewise testable languages and downward closures beyond subwords. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 929–938. ACM, 2018. doi:10.1145/3209108.3209201.